REMARKS/ARGUMENTS

Claims remaining in the present patent application are numbered 1-44. Claims 1, 2, 3, 4, 15, 25, and 35 have been amended. No new matter has been added herein. The rejections and comments of the Examiner set forth in the Office Action dated August 9, 2007 have been carefully considered by the Applicants. Applicants respectfully request the Examiner to consider and allow the remaining claims.

Amendments to the Claims

Claim 1 has been amended to reflect the following:

A method of selecting a media service provider based on static resource information, said method comprising:

identifying a type of service to be performed on an item of content before a service result is provided to a client device, wherein said item of content is identified during a session ~~with~~ between ~~a~~ said client device and a service location manager;

selecting a service provider from a plurality of service providers based on static service provider information and static network information, said selecting of a service provider further based on service session information if said service session information has been received; and

providing information for transferring said session to said service provider, wherein said service provider performs said type of service on said item of content.

Support for the amendment "before a service result is provided to a client device" can be found at least on page 11, second paragraph. Support for the amendment, "during a session between said client device and a service location manager" can be found at least at Figures 1, 2A, 2B, and on page 11, last paragraph. Additionally, the amendment regarding "type of" service is meant to make clear what service to which Applicants are referring.

35 U.S.C. §101 Rejections

Claims 4, and 35-44 are rejected under 35 U.S.C. §101 because the claimed invention is directed to non-statutory subject matter. Applicants respectfully disagree and respectfully traverse the rejection for the following rational.

In *In Re Lowry*, 32 F. 3d 1579,1583-84; 32 USPQ2d 1031, 1035 (Fed. Cir. 1994), the Federal Circuit ruled that an "electronic structure," constructed as a memory containing information stored in a particular arrangement, can serve as the basis for a patentable invention. The Federal Circuit determined that the claimed data structure was a physical entity having specific electrical or magnetic elements in memory. The court considered that the Lowry data structure imposed a physical organization on the data, and found that stored data existing as a collection of bits having information about data relationships may constitute patentable subject matter. Lowry asserted that a memory containing data organized by the claimed data structure permits a computer to efficiently access and to use the stored data, and thus the data structure had tangible benefits.

Applicants respectfully assert that a review of the claims of the instant application against an issued data structure claim of the Lowry patent (e.g., Claim 1 of U.S. Patent No. 5,664,177) demonstrates that the claims of the instant application are statutory.

Also, according to MPEP §2106 Section IV, B. 1, Eighth Edition Incorporating Revision No. 1, "[d]escriptive material can be characterized as either "functional descriptive material" or "nonfunctional descriptive material." In this context, "functional descriptive material" consists

of data structures and computer programs which impart functionality when employed as a computer component. (The definition of 'data structures' is 'a physical or logical relationship among data elements, designed to support specific data manipulation functions.') ... When functional descriptive material is recorded on some computer-readable medium, it becomes structurally and functionally interrelated to the medium and will be statutory in most cases since use of technology permits the function of the descriptive material to be realized."

Furthermore, according to MPEP §2106 Section IV, B. 1. (a),

> a claimed computer-readable medium encoded with a data structure defines structural and functional interrelationships between the data structure and the computer software and hardware components which permit the data structure's functionality to be realized, and is thus statutory.

Emphasis added.

The Supreme Court has held that Congress chose the expansive language of 35 U.S.C. §101 so as to include "anything under the sun that is made by man." *Diamond v. Chakrabarty*, 447 U.S. 303, 308-309; 206 USPQ 193, 197 (1980). Accordingly, 35 U.S.C. §101 provides as follows:

> Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefore, subject to the conditions and requirements of this title.

As cast, 35 U.S.C. §101 defines, as follows, four categories of invention that Congress deems appropriate subject matter of a patent: processes, machines, manufactures, and compositions of matter.

Amended Claim 4 describes:

> A computer useable medium having computer useable code embodied therein for causing a computer to perform operations comprising:
>
> identifying a type of service to be performed on an item of content before a service result is provided to a client device, wherein said item of content is identified during a session between said client device and a service location manager;
>
> selecting a media service provider from a plurality of media service providers based on static service provider information and static network information, said selecting of a service provider further based on service session information if said service session information has been received; and
>
> providing information for transferring said session to said service provider, wherein said service provider performs said type of service on said item of content.

Applicants submit that Claims 4, and 35-44 are directed to a computer useable medium (a machine) for storing computer implemented instructions, said instructions for causing a computer system to perform operations.

Furthermore, MPEP §2106 Section II (C) recites the following:

> Office personnel must always remember to use the perspective of one of ordinary skill in the art. Claims and disclosures are not to be evaluated in a vacuum. If elements of an invention are well known in the art, the applicant does not have to provide a disclosure that describes those elements. In such a case the elements will be construed as encompassing any and every art-recognized hardware or combination of hardware and software technique for implementing the defined requisite functionalities.

> Office personnel are to give claims their broadest reasonable interpretation in light of the supporting disclosure. *In re Morris*, 127 F.3d 1048, 1054-55, 44 USPQ2d 1023, 1027-28 (Fed. Cir. 1997).

Emphasis added. The present Office Action states, "Furthermore, since no explicit definition of "a computer usable medium" is found in the specification, it is a reasonable interpretation to read a computer usable medium as a transmission medium or transmission link such as optical or wired or wireless links."

However, since elements of a computer usable medium are well known in the art, Applicants do not have to provide a disclosure that describes these elements. Additionally, in light of Applicants' supporting disclosure, a reasonable interpretation would not lend itself to interpreting "a computer usable medium" as being "a transmission medium or transmission link such as optical or wired or wireless links", unless a transmission medium or transmission link such as optical or wired or wireless links may have computer useable code embodied therein for causing a computer to perform the operations recited in the Claims (such as Claim 4).

Therefore, Applicants respectfully assert that Claims 4 and 35-44 are directed to statutory subject matter and respectfully request that the rejection be removed.

35 U.S.C. §102(e) Rejections

Claims 1-4, 15-17, and 35-37 are rejected under 35 U.S.C. §102(e) as being anticipated by Menditto et al. (U.S. Patent No. 6,981,029) (Menditto). Applicants have reviewed Menditto and respectfully submit that embodiments of the present invention are neither taught nor suggested by Menditto.

Amended Claim 1 recites:

A method of selecting a media service provider based on static resource information, said method comprising:

identifying a type of service to be performed on an item of content before a service result is provided to a client device, wherein said item of content is identified during a session ~~with~~ between ~~a~~ said client device and a service location manager;

selecting a service provider from a plurality of service providers based on static service provider information and static network information, said selecting of a service provider further based on service session information if said service session information has been received; and

providing information for transferring said session to said service provider, wherein said service provider performs said type of service on said item of content.

Claims 2, 3, 4, 5, 15, and 35 recite similar limitations. Furthermore, Claims 16-17 depend from Claim 15 and recite further limitations, and Claims 36-37 depend from Claim 35 and recite further limitations.

Applicants respectfully submit that Menditto does not teach or suggest, "identifying a type of service to be performed on an item of content before a service result is provided to a client device, wherein said item of content is identified during a session between said client device and a service location manager" as recited in amended Claim 1. Emphasis added.

In Menditto, when a client terminal requests a specific item of content, Menditto's content gateway does not serve the same purpose as Applicants' service location manager. The content gateway does not identify a type of service to be performed on this item of content, and then go through the process of choosing a service provider for performing this type of service on this item of content. In fact, Menditto discloses one or more content gateways within an information service provider (Menditto, Figure 1) which serve the purpose of distributing "information from content providers 14 either directly or through content delivery nodes 22 to

client terminals 16 according to content gateway policy manager 26." Menditto, Col. 2, lines 39-46.

Applicants understand Menditto to focus upon a system which provides for selecting a content provider for delivering requested content in the most efficient manner. Menditto, ABSTRACT, and Col. 1, lines 54-67. Furthermore, Menditto's Col. 1, lines 35-39 state, "...it may be appreciated by those skilled in the art that a need has arisen for a system and technique that can locate an appropriate server to fulfill an information request by using only the contents of the request.".

In contrast, Applicants' invention focuses upon a system which provides for selecting a service provider that can best perform a type of service upon an item of content requested. To select a service provider that may best perform this type of service, Applicants' invention uses more than just the content of a request for an item of content. Applicants' invention, among other information, uses static service provider information, static network information, and service session information if received in order to identify a service provider. Claim 1.

Additionally, there exists a dramatic difference between Applicants' Figure 1 describing Applicants' invention and Menditto's Figure 1 describing Menditto's invention. For example, Applicants' Figure 1 discloses a system in which there is one service location manager 120 and many service providers 130, 132, etc., from which to choose in order to have a type of service performed on content source 110. In contrast, Menditto's Figure 1 discloses a system in which there is one information service provider 12, and many content gateways 18, in order to choose

which content provider 14 to select. As can be seen, Applicants' disclosed system focuses upon choosing a service provider 130, 132, etc., which can best provide a type of service upon an item of content from content source 110. Whereas, Menditto's disclosed system focuses upon choosing a content provider providing a requested item of content.

Therefore, Applicants respectfully assert that the rejection of Claims 1-5, 15-17, and 35-37 under 35 U.S.C. §102(e) is not proper, and that Claims 1-5, 15-17, and 35-37 thus overcome the rejection under 35 U.S.C. §102(e).

35 U.S.C. §103(a) Rejections

The present Office Action rejected Claims 5-14, 18-34, and 38-44 under 35 U.S.C. §103(a) as being unpatentable over Menditto in view of Bochmann et al. (Quality of service management issues in electronic commerce applications) (hereinafter, Bochmann). Applicants respectfully submit that embodiments of the present invention are neither taught nor suggested by Menditto or Bochmann, alone or in combination.

Bochmann does not remedy the deficiencies in Menditto in that neither Bochmann nor Menditto teach or suggest "identifying a type of service to be performed on an item of streaming content before a service result is provided to a client device, wherein said item of streaming content is identified during a session between said client device and a service location manager", as recited by Claim 5. Independent Claims 5, 15, 25, and 35 should be patentable for similar reasons stated herein that independent Claim 1 should be patentable. Claims 6-14 depend on Claim 5. Claims 18-24 depend from Claim 15. Claims 26-34 depend from Claim 25. Claims

38-44 depend from Claim 35. These dependent claims should be patentable for at least the reasons that their respective independent claims should be patentable.

CONCLUSION

In light of the amendments and remarks presented herein, Applicants respectfully request reconsideration of the rejected Claims for allowance thereof.

Based on the arguments presented above, Applicants respectfully assert that Claims 1-44 overcome the rejections of record. Therefore, Applicants respectfully solicit allowance of these Claims.

The Examiner is urged to contact Applicants' undersigned representative if the Examiner believes such action would expedite resolution of the present Application.

Respectfully submitted,
WAGNER BLECHER

Date: 10/9, 2007

John P. Wagner
Reg. No. 35,398
123 Westridge Dr.
Watsonville, CA 95076

Architectures for MPEG Compressed Bitstream Scaling

Huifang Sun, *Senior Member, IEEE,* Wilson Kwok, *Member, IEEE,* and Joel W. Zdepski

Abstract—**The idea of moving picture expert group (MPEG) bitstream scaling relates to altering or scaling the amount of data in a previously compressed MPEG bitstream. The new scaled bitstream conforms to constraints that are not known nor considered when the original precoded bitstream was constructed. Numerous applications for video transmission and storage are being developed based on the MPEG video coding standard. Applications such as video on demand, trick-play track on digital video tape recorders (VTR's) and extended-play recording on VTR's motivate the idea of bitstream scaling. In this paper, we present several bitstream scaling methods for the purpose of reducing the rate of constant bit rate (CBR) encoded bitstreams. The different methods have varying hardware implementation complexity and associated trade-offs in resulting image quality. Simulation results on MPEG test sequences demonstrate the typical performance trade-offs of the methods.**

I. Introduction

DIGITAL video signal processing is an area of science and engineering that has developed rapidly over the past decade. The maturity of the moving picture expert group (MPEG) video coding standard [1] is a very important achievement for the video industry and provides a strong support for digital transmission and storage of video signals. The MPEG coding standard is now being deployed for a variety of applications which include high definition television (HDTV), teleconferencing, direct broadcasting by satellite (DBS), interactive multimedia terminals, and digital video disc.

The basic function of a bitstream scaler may be thought of as a black box which passively accepts a precoded MPEG bitstream at the input and produces a scaled bitstream which meets new constraints that are not known *a priori* during the creation of the original precoded bitstream. The bitstream scaler is a transcoder, or filter, that provides a match between an MPEG source bitstream and the receiving load. The receiving load consists of the transmission channel, the destination decoder, and perhaps a destination storage device. For example, the constraint on the new bitstream may be a bound on the peak or average bit rate imposed by the communications channel, or a bound on the total number of bits imposed by the storage device, or a bound on the variation of bit usage across pictures due to the amount of buffering available at the receiving decoder.

While the idea of bitstream scaling shares many similar concepts as those provided by the various MPEG-2 scalability profiles, the intended applications and goals differ. The MPEG-2 scalability methods (data partitioning, SNR scalability, spatial scalability, and temporal scalability) are aimed at providing encoding of source video into multiple service grades (that are predefined at the time of encoding) and multitiered transmission for increased signal robustness. The multiple bitstreams created by MPEG-2 scalability are hierarchically dependent in such a way that by decoding an increasing number of bitstreams, higher service grades are reconstructed. Bitstream scaling methods, in contrast, are primarily decoder/transcoder techniques for converting an existing precoded bitstream to another one that meets new rate constraints. Several applications that motivate bitstream scaling include:

i) Video On-Demand—Consider a video on-demand (VOD) scenario wherein a video file-server includes a storage device containing a library of precoded MPEG bitstreams. These bitstreams in the library are originally coded at a high quality (e.g., studio quality). A number of clients may request retrieval of these video programs at one particular time. The number of users and the quality of video delivered to the users is constrained by the outgoing channel capacity. This outgoing channel, which may be a cable bus or an ATM trunk, for example, must be shared among the users who are admitted service. Different users may require different levels of video quality, and the quality of a respective program will be based on the fraction of the total channel capacity allocated to each user. To simultaneously accommodate a plurality of users, the video file-server must scale the stored precoded bitstreams to a reduced rate before it is delivered over the channel to the respective users. The quality of the resulting scaled bitstream should not be significantly degraded compared to the quality of a hypothetical bitstream so obtained by coding the original source material at the reduced rate. Complexity cost is not such a critical factor because only the file-server has to be equipped with the bitstream scaling hardware, not every user. Presumably, video service providers would be willing to pay a high cost for delivering close to the highest quality video possible at a prescribed bit rate.

As an option, a sophisticated video file-server may also perform scaling of multiple original precoded bitstreams jointly and statistically multiplex the resulting

Manuscript received July 3, 1995; revised December 13, 1995. This paper was recommended by Associate Editor H. Gharavi.

H. Sun was with the David Sarnoff Research Center, Princeton, NJ USA. He is now with ATV Laboratory, Mitsubishi Electric Information Technology Center America, Somerset, NJ 08873 USA.

W. Kwok was with the David Sarnoff Research Center, Princeton, NJ USA. He is now with C-Cube Microelectronics, Milpitas, CA 95053 USA.

J. Zdepski was with the David Sarnoff Research Center, Princeton, NJ USA. He is now with Thomson Sun Interactive Alliance, Mountain View, CA 94043-1100 USA.

Publisher Item Identifier S 1051-8215(96)03019-4.

scaled variable bit rate (VBR) bitstreams into the channel. By scaling the group of bitstreams jointly, statistical gains can be achieved. These statistical gains can be realized in the form of higher and more uniform picture quality for the same channel capacity. Statistical multiplexing over a DirecTv transponder [2] is one example application of video statistical multiplexing.

ii) Trick-Play Track on Digital VTR's—In this application, the video bitstream is scaled to create a sidetrack on video tape recorders. This side track contains very coarse quality video sufficient to facilitate trick-modes on the video tape recorder (VTR) (e.g., fast forward and rewind at different speeds). Complexity cost for the bitstream scaling hardware is of significant concern in this application since the VTR is a mass consumer item subject to mass production.

iii) Extended-Play Recording on Digital VTR's—In this application, video is broadcast to users' homes at a certain broadcast quality (~6 Mbps for standard definition video and ~24 Mbps for high definition video). With a bitstream scaling feature in their video tape recorders, users may record the video at a reduced rate, akin to extended play (EP) mode on today's VHS recorders, thereby recording a greater duration of video programs onto a tape at lower quality. Again, hardware complexity costs would be a major factor here.

In this paper we limit ourselves to focus on the problem of scaling an MPEG constant bit rate (CBR) encoded bitstream down to a lower constant bit rate. The rest of this paper is organized as follows. The basic principles of bitstream scaling are described in the Section II. Several basic architectures for bitstream scaling are presented in Section III. The simulation results and concluding remarks are given in Sections IV and V, respectively.

II. Basic Principles of Bitstream Scaling

As described previously, the idea of scaling an MPEG-2 compressed bitstream down to a lower bit rate is initiated by several applications. The problem arises here is what criteria can be used to judge the performances of an architecture which can reduce the size or rate of a MPEG compressed bitstream. Two basic principles of bitstream scaling we used are: 1) the information in the original bitstream should be exploited as much as possible, and 2) the resulting image quality of the new bitstream with a lower bit rate should be as high as possible, or as close as possible to a bitstream created by coding the original source video at the reduced rate. Here, we assume that for a given rate, the original source is encoded in an optimal way. Of course, the implementation of hardware complexity has also to be considered. Fig. 1 shows a simplified encoding structure of an MPEG encoding in which the rate control mechanism is not shown.

In this structure, a block of image data is first transformed to a set of coefficients, the coefficients are then quantized with a quantizer step which is decided by the given bit rate budget or number of bits assigned to this block. Finally, the quantized coefficients are coded in variable length coding to the binary format, called bitstream or bits.

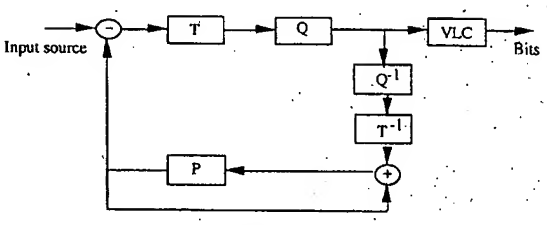


Fig. 1. Simplified encoder structure.

From this structure it is obvious that the performance of changing the quantizer step will be better than cutting higher frequencies when the same amount of rate needs to be reduced. In the original bitstream, the coefficients are quantized with finer quatization steps which are optimized at the original high rate. After cutting the coefficients of higher frequencies, the rest of the coefficients are not quantized with an optimal quantizer. In the method of requantization, all coefficients are requantized with an optimal quantizer which is determined by the reduced rate. The performance of this method must be better than one by cutting high frequencies to reach the reduced rate. We give the theoretical analysis in the Appendix.

In this paper, several different architectures have been proposed for the bitstream scaling using these two methods with other issues related to the features of video signals, such as motion compensation. The different methods have varying hardware implementation complexity, with each having its own degree of trade-off between required hardware and resulting image quality.

III. Architectures of Bitstream Scaling

Four architectures for bitstream scaling are proposed. Each of the scaling architectures described has its own particular benefits that are suitable for particular applications.

Architecture 1: The bitstream is scaled by cutting high frequencies.

Architecture 2: The bitstream is scaled by requantization.

Architecture 3: The bitstream is scaled by re-encoding the reconstructed pictures with motion vectors and coding decision modes extracted from the original high-quality bitstream.

Architecture 4: The bitstream is scaled by re-encoding the reconstructed pictures with motion vectors extracted from the original high-quality bitstream, but new coding decisions are computed based on reconstructed pictures.

Architectures 1 and 2 are considered for VTR applications such as trick-play modes and EP recording. Architectures 3 and 4 are considered for video on-demand and other applicable StatMux scenarios.

A. Architecture 1: Cutting AC Coefficients

A block diagram illustrating Architecture 1 is shown in Fig. 2(a). The method of reducing the bit rate in Architecture 1 is based on cutting the higher frequency coefficients. The incoming precoded constant bit rate stream enters a decoder rate buffer. Following the top branch leading from the rate buffer, a variable lenght decoder (VLD) is used to parse the

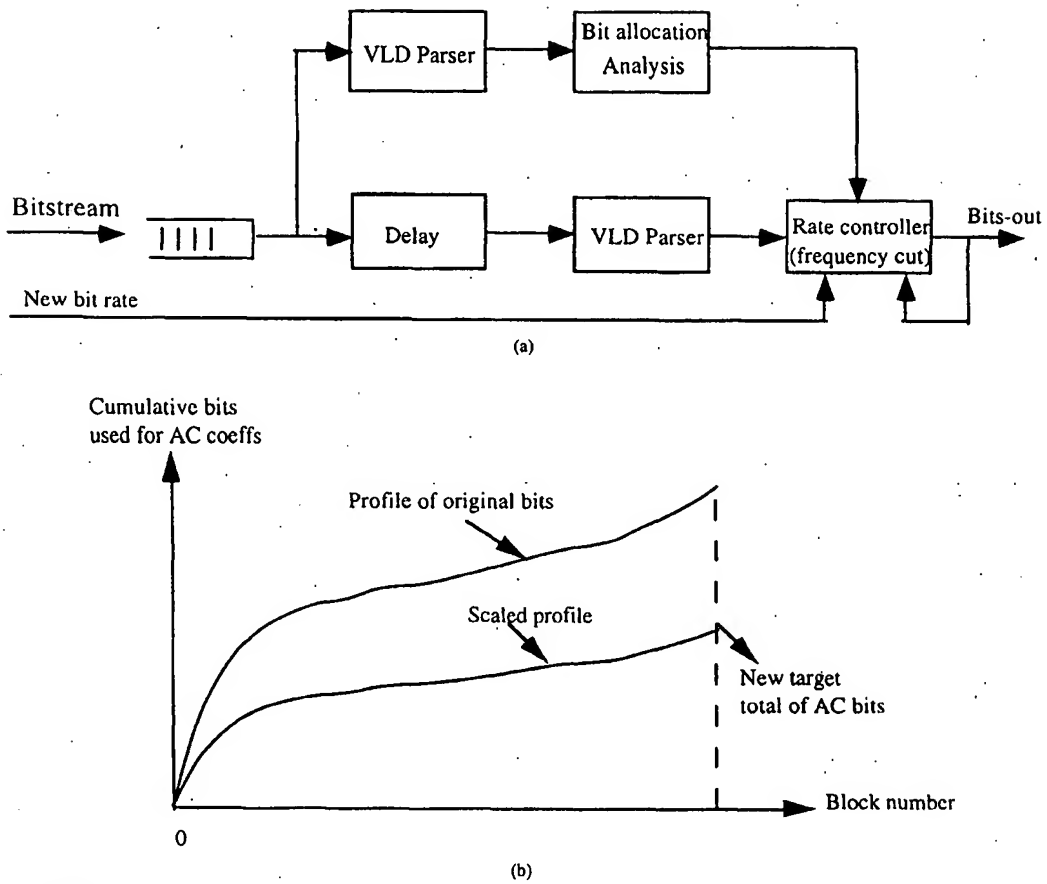


Fig. 2. (a) Architecture 1: Cutting high frequencies. (b) Profile map.

bits for the next frame in the buffer to identify all the variable length codewords that correspond to ac coefficients used in that frame. No bits are removed from the rate buffer. The codewords are not decoded, but just simply parsed by the VLD parser to determine codeword lengths. The bit allocation analyzer accumulates these ac bit-counts for every macroblock in the frame and creates an ac bit usage profile as shown in Fig. 2(b). That is, the analyzer generates a running sum of ac digital cosine transformation (DCT) coefficients bits on a macroblock basis

$$\mathrm{PV}_N = \sum \mathrm{AC\_BITS} \tag{1}$$

where PV_N is the profile value of a running sum of ac codeword bits until to macroblock N.

In addition, the analyzer counts the sum of all coded bits for the frame, TB (total bits). After all macroblocks for the frame have been analyzed, a target value $\mathrm{TV}_{\mathrm{AC}}$, of ac DCT coefficient bits per frame is calculated as

$$\mathrm{TV}_{\mathrm{AC}} = \mathrm{PV}_{\mathrm{LS}} - \alpha^{*}\mathrm{TB} - B_{\mathrm{EX}} \tag{2}$$

where $\mathrm{TV}_{\mathrm{AC}}$ is the target value of ac DCT codeword bits per frame, $\mathrm{PV}_{\mathrm{LS}}$ is the profile value at the last macroblock, α is the percentage by which the precoded bitstream is to be reduced, TB is the total bits and B_{EX} is the amount of bits by which the previous frame missed its desired target.

The profile value of ac DCT coefficient bits is scaled by the factor $\mathrm{TV}_{\mathrm{AC}}/\mathrm{PV}_{\mathrm{LS}}$. Scaling is performed by multiplying each PV_N by that factor to generate the linearly scaled profile shown in Fig. 2(b). Following the bottom branch from the rate buffer, a delay is inserted equal to the amount of time required for the top branch analysis processing to be completed for the current frame. A second VLD parser accesses and removes all codeword bits from the buffer and delivers them to a rate controller. The rate controller receives the scaled target bit usage profile for the amount of ac bits to be used within the frame. The rate controller has memory to store all coefficients associated with the current macroblock it is operating on. All original codeword bits at a higher level than ac coefficients (i.e., all fixed length header codes, motion vector codes, macroblock type codes, etc.) are held in memory and will be remultiplexed with all ac DCT codewords in that macroblock that have not been excised to form the outgoing scaled bitstream. The rate controller determines and flags in the macroblock codeword memory which ac DCT codewords to keep and which to excise. Ac DCT codewords are accessed from the macroblock codeword memory in the order AC11, AC12, AC13, AC14, AC15, AC16, AC21, AC22,

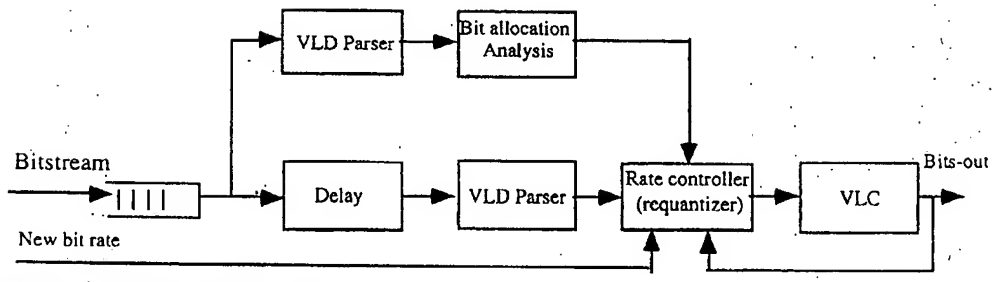


Fig. 3. Architecture 2: Increasing quantization step.

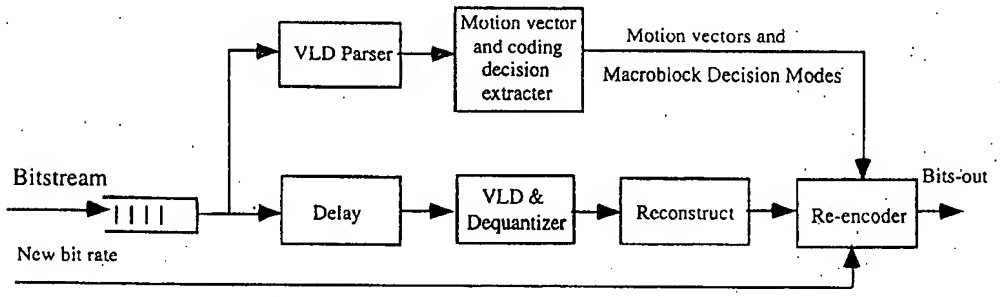


Fig. 4. Architecture 3: Re-encoding at lower bit rate with motion vectors and macroblock coding decisions extracted from the original bits.

AC23, AC24, AC25, AC26, AC31, AC32, AC33, etc., where AC_{ij} denotes the ith ac DCT codewords from jth block in the macroblock if it is present. As the ac DCT codewords are accessed from memory, the respective codeword bits are summed and continuously compared with the scaled profile value to the current macroblock, less number of bits for insertion of end-of-block (EOB) codewords. Respective ac DCT codewords are flagged as kept until the running sum of ac DCT codewords bits exceed the scaled profile value less EOB bits. When this condition is met, all remaining ac DCT codewords are marked for being excised. This process continues until all macroblocks have their kept codewords reassembled to form the scaled bitstream.

B. Architecture 2: Increasing Quantization Step

Architecture 2 is shown in Fig. 3.

The method of bitstream scaling in Architecture 2 is based on increasing the quantization step. This method requires additional dequantizer/quantizer and variable length coding (VLC) hardware over the first method. Like the first method, it also makes a first VLD pass on the bitstream and obtains a similar scaled profile of target cumulative codeword bits versus macroblock count to be used for rate control.

The rate control mechanism differs from this point on. After the second pass VLD is made on the bitstream, quantized DCT coefficients are dequantized. A block of finely quantized DCT coefficients is obtained as a result of this. This block of DCT coefficients is requantized with a coarser quantizer scale. The value used for the coarser quantizer scale is determined adaptively by making adjustments after every macroblock so that the scaled target profile is tracked as we progress through the macroblocks in the frame

$$Q_N = Q_{\mathrm{NOM}} + G * \left(\sum_{N-1} \mathrm{BU} - \mathrm{PV}_{N-1} \right) \tag{3}$$

where Q_N is the quantization factor for macroblock N, Q_{NOM} is an estimate of the new nominal quantization factor for the frame, $\sum_{N-1}$ BU is the cumulative amount of coded bits up to macroblock $N-1$, and G is a gain factor which controls how tightly the profile curve is tracked through the picture. Q_{NOM} is initialized to an average guess value before the very first frame, and updated for the next frame by setting it to Q_{LS} (the quantization factor for the last macroblock) from the frame just completed. The coarsely requantized block of DCT coefficients is variable-length-coded to generate the scaled bitstream. The rate controller also has provisions for changing some macroblock-layer codewords, such as the macroblock-type and coded-block-pattern to ensure a legitimate scaled bitstream that conforms to MPEG-2 syntax.

C. Architecture 3: Re-Encoding with Old Motion Vectors and Old Decisions

The third architecture for bitstream scaling is shown in Fig. 4.

In this architecture, the motion vectors and macroblock coding decision modes are first extracted from the original bitstream, and at the same time the reconstructed pictures are obtained from the normal decoding procedure. Then the scaled bitstream is obtained by re-encoding the reconstructed pictures using the old motion vectors and macroblock decisions from the original bitstream. The benefits obtained from this architecture compared to full decoding and re-encoding is that no motion estimation and decision computation is needed.

TABLE I
HARDWARE COMPLEXITY SAVINGS OVER FULL DECODING/RE-ENCODING

Coding method	Hardware Complexity Savings
Architecture 1	No decoding loop, no DCT/IDCT, no frame store memory, no encoding loop, no quantizer/dequantizer, no motion compensation, no VLC, simplified rate control.
Architecture 2	No decoding loop, no DCT/IDCT, no frame store memory, no encoding loop, no motion compensation, simplified rate control.
Architecture 3	No motion estimation, no macroblock coding decisions.
Architecture 4	No motion estimation.

D. Architecture 4: Re-Encoding with Old Motion Vectors and New Decisions

Architecture 4 is a modified version of Architecture 3 in which new macroblock decision modes are computed during re-encoding based on reconstructed pictures. The scaled bitstream created this way is expected to yield an improvement in picture quality because the decision modes obtained from the high-quality original bitstream are not optimum for re-encoding at the reduced rate. For example, at higher rates, the optimal mode decision for a macroblock is more likely to favor bidirectional field motion compensation over forward frame motion compensation. But at lower rates, just the opposite decision may be true. In order for the re-encoder to have the possibility of deciding on new macroblock coding modes, the entire pool of motion vectors of every type must be available. This can be supplied by augmenting the original high-quality bitstream with ancillary data containing the entire pool of motion vectors during the time it was originally encoded. It could be inserted into user data every frame. For the same original bit rate, the quality of an original bitstream obtained this way is degraded compared to an original bitstream obtained from Architecture 3 because the additional overhead required for the extra motion vectors steals away bits for actual encoding. However, the resulting scaled bitstream is expected to show quality improvement over the scaled bitstream from Architecture 3 if the gains from computing new and more accurate decision modes can overcome the loss in original picture quality.

Table I outlines the hardware complexity savings of each of the three proposed architectures as compared to full decoding and re-encoding.

IV. SIMULATION RESULTS

The four proposed bitstream scaling architectures are evaluated on MPEG-2 Test Model 4 precoded bitstreams.

In the first set of simulation results (Table II), MPEG committee test sequences "Flower Garden," "Bicycle," and "Bus" are used. Each test sequence is 150 frames at CCIR-601 720 × 480 4 : 2 : 0 resolution. The original precoded bitstream is generated with a high bit rate (15 Mbps) and then its size is scaled by 73% to a much lower bit rate (4 Mbps) by using the four proposed architectures. We tabulate eight coding methods for each test sequence:

TABLE II
15 MBPS SCALED DOWN TO 4 MBPS BITSTREAM SCALING EXPERIMENT

Coding Method	PSNR (dB)		
	Flower garden	Bicycle	Bus
Method 1	37.02	35.12	37.74
Method 2	35.77	33.87	36.69
Method 3	29.60	27.26	30.44
Method 4	29.34	27.14	30.22
Method 5	20.89	20.18	21.87
Method 6	27.41	25.04	28.44
Method 7	29.19	27.02	30.14
Method 8	29.32	27.11	30.20

TABLE III
4 MBPS ⇒ 3.2 MBPS BITSTREAM SCALING EXPERIMENT

Coding method	Bit rate (Mbps)	PSNR (dB)
Coding original source	4.0	29.60
Coding original source	3.2	28.31
Architecture 1	3.2	25.82
Architecture 2	3.2	26.18
Architecture 3	3.2	27.30

Method 1: The original precoded bitstream at 15 Mbps.
Method 2: The original precoded bitstream at 15 Mbps augmented with all motion vectors in user data (effectively about 12 Mbps for actual encoding, 3 Mbps for extraneous motion vectors).
Method 3: The original source video coded at 4 Mbps.
Method 4: Bitstream scaling by doing full decoding of the precode 15 Mbps bitstream and full re-encoding at 4 Mbps using Test Model 4.
Method 5: Scaled bitstream using Architecture 1.
Method 6: Scaled bitstream using Architecture 2.
Method 7: Scaled bitstream using Architecture 3.
Method 8: Scaled bitstream using Architecture 4.

Methods 1 through 4 serve as references for comparing the scaling results.

In the second simulation (Table III), the original precoded "Flower Garden" bitstream is obtained with a low bit rate (4 Mbps). Then the size of bitstream is reduced by a smaller percentage, 20% in this case, which is not as significant as in the first simulation set. Note that Architecture 4 is not applicable for this scenario where the precoded bit rate is only 4.0 Mbps (there is not enough room at that rate for all the additional motion vectors).

We notice from the 15 Mbps ⇒ 4 Mbps scaling experiment (Table II), that Architecture 3 produces a bitstream that is degraded by 0.25 to 0.4 dB compared to coding the original source material at 4 Mbps. Architecture 4's original bitstream suffers around a 1.3 dB drop at 15 Mbps but yields a scaled bitstream at 4 Mbps that is 0.06 to 0.13 dB higher than Architecture 3's scaled bitstream. For the 4 Mbps ⇒ 3.2 Mbps experiments, we notice that the open-loop scaling methods of Architectures 1 and 2 perform much better than scaling from a high bit rate precoded source.

Fig. 5 shows the pictorial results for a reconstructed frame corresponding to the numerical results shown in Table III. The particular frame is number 27; this frame is the last coded P-frame in a GOP. Fig. 5(a) and (b) are the original source

(a)

(b)

Fig. 5. Pictorial results for a reconstructed frame of the "Flower Garden" sequence. Recontructed from (a) the original bitstream at the bit rate of 4 Mbps and (b) the original bitstream at 3.2 Mbps.

coded at the bit rate of 4 Mbps and 3.2 Mbps, respectively. Fig. 5(c) (d), and (e) are reconstructed pictures from the scaled bitstream at the bit rate of 3.2 Mbps with Architectures 1, 2, and 3, respectively. The worst case drift errors incurred under Architectures 1 and 2 can be seen from Fig. 5(c) and (d).

V. Conclusion

We have proposed some architectures for bitstream scaling which are useful for various applications as described in the introduction. Among the four architectures, Architectures 1 and 2 do not require entire decoding and encoding loops nor frame store memory for reconstructed pictures, thereby saving significant hardware complexity. However, video quality tends to degrade through the group-of-pictures until the next I-picture due to drift in the absence of decoder/encoder loops. For large scaling, say for rate reduction greater than 25%, Architecture 1 produces poor-quality blocky pictures, primarily because many bits were spent in the original high-

(c)

(d)

Fig. 5. *(Continued.)* Pictorial results for a reconstructed frame of the "Flower Garden" sequence. Recontructed from (c) the scaled bitstream at 3.2 Mbps with Architecture 1 and (d) the scaled bitstream at 3.2 MBPS with Architecture 2.

quality bitstream on finely quantizing the dc and other very low-order ac coefficients. Architecture 2 is a particularly good choice for VTR applications since it is a good compromise between the hardware complexity and reconstructed image quality. Architectures 3 and 4 are suitable for VOD server applications and other StatMux applications.

APPENDIX

In this analysis, we assume that the optimal quantizer is obtained by assigning the number of bits according to the variance or energy of the coefficients. It is slightly different from the MPEG standard, which will be explained later, but the principal concept is the same and the results will hold for the MPEG standard. We first analyze the errors caused by cutting high coefficients and increasing the quantizer step. The optimal bit assignment is given by [3]

$$R_{k0} = R_{av0} + \frac{1}{2}\log_2 \frac{\sigma_k^2}{\left(\prod_{i=0}^{N-1}\sigma_i^2\right)^{1/N}}, \quad k = 0, 1, \cdots, N-1 \qquad (4)$$

(e)

Fig. 5. *(Continued.)* Pictorial results for a reconstructed frame of the "Flower Garden" sequence. Recontructed from (e) the scaled bitstream at 3.2 Mbps with Architecture 3.

where N is the number of coefficients in the block, R_{k0} is the number of bits assigned to the kth coefficient, R_{av0} is the average number of bits assigned to each coefficient in the block, i.e., $R_{T0} = N \cdot R_{av0}$, is the total bits for this block under a certain bit rate and σ_k^2 is the variance of kth coefficient. Under the optimal bit assignment (4), the minimized average quantizer error, σ_{q0}^2, is

$$\sigma_{q0}^2 = \frac{1}{N}\sum_{k=1}^{N-1}\sigma_{qk}^2 = \frac{1}{N}\sum_{k=1}^{N-1} 2^{-2R_{k0}} \cdot \sigma_k^2 \tag{5}$$

where σ_{qk}^2 is the quantizer error of the kth coefficient. According to (4), we have two major methods to reduce the bit rate, either cutting high coefficients or decreasing the R_{av}, i.e., increasing the quantizer step. We are now analyzing the affects on the reconstructed errors caused due to the cut of bits using these methods. Assume that the number of the bits assigned to the block is reduced from R_{T0} to R_{T1}. Then the bits to be reduced, ΔR_1 are equal to $R_{T0} - R_{T1}$.

In the case of cutting high frequencies, say the number of coefficients is reduced from N to M, then

$$R_{k0} = 0 \quad \text{for } K \geq M,$$

and

$$\Delta R_1 = R_{T0} - R_{T1} = \sum_{k=M}^{N-1} R_{k0}$$

the quantizer error increased due to the cutting is

$$\begin{aligned}\Delta\sigma_{q1}^2 &= \sigma_{q1}^2 - \sigma_{q0}^2 \\ &= \frac{1}{N}\left(\sum_{k=0}^{M-1} 2^{-2R_{k0}} \cdot \sigma_k^2 + \sum_{k=M}^{N-1}\sigma_k^2 - \sum_{k=0}^{N-1} 2^{-2R_{k0}} \cdot \sigma_k^2\right) \\ &= \frac{1}{N}\left(\sum_{k=M}^{N-1}\sigma_k^2 - \sum_{k=M}^{N-1} 2^{-2R_{k0}} \cdot \sigma_k^2\right) \\ &= \frac{1}{N}\sum_{k=M}^{N-1}(1 - 2^{-2R_{k0}}) \cdot \sigma_k^2 \end{aligned} \tag{6}$$

where σ_{q1}^2 is the quantizer error after cutting the high frequencies.

In the method of increasing quantizer step, or decreasing the average bits, from R_{av0} to R_{av2}, assigned to each coefficient, the number of bits reduced for the block is

$$\Delta R_2 = R_{T0} - R_{T2} = N \cdot (R_{av0} - R_{av2})$$

and the bits assigned to each coefficient become now

$$R_{k2} = R_{av2} + \frac{1}{2}\log_2 \frac{\sigma_k^2}{\left(\prod_{i=0}^{N-1}\sigma_i^2\right)^{1/N}}, \quad k = 0, 1, \ldots, N-1. \tag{7}$$

The corresponding quantizer error increased by the cutting bits is

$$\begin{aligned}\Delta\sigma_{q2}^2 &= \sigma_{q2}^2 - \sigma_{q0}^2 \\ &= \frac{1}{N}\left(\sum_{k=0}^{N-1} 2^{-2R_{k2}} \cdot \sigma_k^2 - \sum_{k=0}^{N-1} 2^{-2R_{k0}} \cdot \sigma_k^2\right) \\ &= \frac{1}{N}\sum_{k=0}^{N-1}(2^{-2R_{k2}} - 2^{-2R_{k0}}) \cdot \sigma_k^2 \end{aligned} \tag{8}$$

where σ_{q2}^2 is the quantizer error at the reduced bit rate. If the same number of bits is reduced, i.e., $\Delta R_1 = \Delta R_2$, it is obvious that $\Delta\sigma_{q2}^2$ is smaller than $\Delta\sigma_{q1}^2$ since σ_{q2}^2 is the minimized value at the reduced rate. This implies that the

performance of changing the quantizer step will be better than cutting higher frequencies when the same amount of rate needs to be reduced. It should be noted that in the MPEG video coding, the more sophisticated bit assignment algorithms are used. First, different quantizer matrices are used to improve the visual perceptual performance. Second, different VLC tables are used to code the dc values and the ac transform coefficients and the run length coding is used to code the pairs of the zero-run length and the values of amplitudes. However, in general, the bits are still assigned according to the statistical model which indicates the energy distribution of the transform coefficients. Therefore, the above theoretical analysis will hold for the MPEG video coding.

ACKNOWLEDGMENT

The authors wish to thank R. Kalluri of Sun/Thomson Interactive and T. Chiang of the David Sarnoff Research Center for facilitating the simulation results obtained in this paper.

REFERENCES

[1] MPEG-2 International Standard, *Video Recommendation ITU-T H.262*, ISO/IEC 13818-2, Jan. 20, 1995.
[2] M. Perkins and D. Arnstein, "Statistical multiplexing of multiple MPEG-2 video programs in a single channel," *SMPTE J.*, pp. 596–599, Sept. 1995.
[3] N. N. Jayant and P. Noll, *Digital Coding of Waveforms to Speech and Video*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Huifang Sun (S'83–M'85–SM'93) received the B.S. degree in electrical engineering from Harbin Engineering Institute, Harbin, China, in 1967, and the Ph.D. degree in electrical engineering from University of Ottawa, Ottawa, Canada, in 1986.

In 1970 he took a position as an Engineer at the Shanghai Aeronautical Radio and Electronic Research Institute, Shanghai, China. In 1981, he was a visitor at Laval University, Quebec, Canada. From 1982 to 1986, he was with electrical engineering department at University of Ottawa, Ottawa, Canada. In 1986, he joined Fairleigh Dickinson University, Teaneck, NJ, as an Assistant Professor and consequently promoted to an Associate Professor in electrical engineering. From 1990 to 1995 he was with the David Sarnoff Research Center (formerly RCA Lab), Princeton, NJ, as a Member of Technical Staff and consequently promoted to Technology Leader of Digital Video Technology where his activities were MPEG video coding, AD-HDTV, and Grand Alliance HDTV development. He joined the Advanced Television Laboratory, Mitsubishi Electric Information Technology Center America (ITA), Somerset, NJ, in 1995 as a Senior Principal Technical Staff where his activity is advanced television development. He holds two U.S. patents and has several pending, and has authored or coauthored more than 50 journal and conference papers.

Wilson Kwok (M'93) was born in Brooklyn, NY, in 1970. He received his electrical engineering Masters degree from The Cooper Union, New York City, in 1992.

Upon graduation, he joined The David Sarnoff Research Center in Princeton, NJ, where he got a post graduate education in digital video and became a Member of the Technical Staff. Since 1995 he has worked for C-Cube Microsystems, Milpitas, CA, where he now works in the algorithms research group. His recreational interests include cycling, hiking, and playing stocks. He holds four issued U.S. patents, with several pending, and has authored or coauthored more than 10 journal and conference papers.

Joel W. Zdepski received the B.S.E.E., M.S.E.E., and Ph.D. degrees from Rutgers University, New Brunswick, NJ.

From 1981 to 1984 he was employed by Dranetz Technologies Inc. In 1986 he joined the Communications Laboratory at the David Sarnoff Research Center, where he concentrated on video compression and video communications technology, particularly video teleconferencing, digital television, and high definition television. He contributed to the development of AD-HDTV and the Digital Satellite System (DSS). Mr. Zdepski was member of the MPEG-2 Video and Systems committees. He was also a member of both the Video Compression and Transport Specialist Groups within the HDTV consortium know as the Grand Alliance. In 1995, he joined the Thomson Sun Interactive Alliance, Mountain View, CA, as Manager of the Servers and Communications group, where his activity is interactive television. He holds 12 issued patents, with several pending, and has authored or coauthored more than 20 journal and conference papers.

Network Working Group
Request for Comments: 1889
Category: Standards Track

Audio-Video Transport Working Group
H. Schulzrinne
GMD Fokus
S. Casner
Precept Software, Inc.
R. Frederick
Xerox Palo Alto Research Center
V. Jacobson
Lawrence Berkeley National Laboratory
January 1996

RTP: A Transport Protocol for Real-Time Applications

Status of this Memo

Abstract

This memorandum describes RTP, the real-time transport protocol. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers. The protocol supports the use of RTP-level translators and mixers.

Table of Contents

1. Introduction

This memorandum specifies the real-time transport protocol (RTP), which provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, timestamping and delivery monitoring. Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services; both protocols contribute parts of the transport protocol functionality. However, RTP may be used with other suitable underlying network or transport protocols (see Section 10). RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network.

Note that RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do so. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence.

While RTP is primarily designed to satisfy the needs of multi-participant multimedia conferences, it is not limited to that particular application. Storage of continuous data, interactive distributed simulation, active badge, and control and measurement applications may also find RTP applicable.

This document defines RTP, consisting of two closely-linked parts:

- o the real-time transport protocol (RTP), to carry data that has real-time properties.

- o the RTP control protocol (RTCP), to monitor the quality of service and to convey information about the participants in an on-going session. The latter aspect of RTCP may be sufficient for "loosely controlled" sessions, i.e., where there is no explicit membership control and set-up, but it is not necessarily intended to support all of an application's control communication requirements. This functionality may be fully or partially subsumed by a separate session control protocol,

which is beyond the scope of this document.

RTP represents a new style of protocol following the principles of application level framing and integrated layer processing proposed by Clark and Tennenhouse [1]. That is, RTP is intended to be malleable to provide the information required by a particular application and will often be integrated into the application processing rather than being implemented as a separate layer. RTP is a protocol framework that is deliberately not complete. This document specifies those functions expected to be common across all the applications for which RTP would be appropriate. Unlike conventional protocols in which additional functions might be accommodated by making the protocol more general or by adding an option mechanism that would require parsing, RTP is intended to be tailored through modifications and/or additions to the headers as needed. Examples are given in Sections 5.3 and 6.3.3.

Therefore, in addition to this document, a complete specification of RTP for a particular application will require one or more companion documents (see Section 12):

- o a profile specification document, which defines a set of payload type codes and their mapping to payload formats (e.g., media encodings). A profile may also define extensions or modifications to RTP that are specific to a particular class of applications. Typically an application will operate under only one profile. A profile for audio and video data may be found in the companion RFC TBD.

- o payload format specification documents, which define how a particular payload, such as an audio or video encoding, is to be carried in RTP.

A discussion of real-time services and algorithms for their implementation as well as background discussion on some of the RTP design decisions can be found in [2].

Several RTP applications, both experimental and commercial, have already been implemented from draft specifications. These applications include audio and video tools along with diagnostic tools such as traffic monitors. Users of these tools number in the thousands. However, the current Internet cannot yet support the full potential demand for real-time services. High-bandwidth services using RTP, such as video, can potentially seriously degrade the quality of service of other network services. Thus, implementors should take appropriate precautions to limit accidental bandwidth usage. Application documentation should clearly outline the limitations and possible operational impact of high-bandwidth real-

time services on the Internet and other network services.

2. RTP Use Scenarios

The following sections describe some aspects of the use of RTP. The examples were chosen to illustrate the basic operation of applications using RTP, not to limit what RTP may be used for. In these examples, RTP is carried on top of IP and UDP, and follows the conventions established by the profile for audio and video specified in the companion Internet-Draft draft-ietf-avt-profile

2.1 Simple Multicast Audio Conference

A working group of the IETF meets to discuss the latest protocol draft, using the IP multicast services of the Internet for voice communications. Through some allocation mechanism the working group chair obtains a multicast group address and pair of ports. One port is used for audio data, and the other is used for control (RTCP) packets. This address and port information is distributed to the intended participants. If privacy is desired, the data and control packets may be encrypted as specified in Section 9.1, in which case an encryption key must also be generated and distributed. The exact details of these allocation and distribution mechanisms are beyond the scope of RTP.

The audio conferencing application used by each conference participant sends audio data in small chunks of, say, 20 ms duration. Each chunk of audio data is preceded by an RTP header; RTP header and data are in turn contained in a UDP packet. The RTP header indicates what type of audio encoding (such as PCM, ADPCM or LPC) is contained in each packet so that senders can change the encoding during a conference, for example, to accommodate a new participant that is connected through a low-bandwidth link or react to indications of network congestion.

The Internet, like other packet networks, occasionally loses and reorders packets and delays them by variable amounts of time. To cope with these impairments, the RTP header contains timing information and a sequence number that allow the receivers to reconstruct the timing produced by the source, so that in this example, chunks of audio are contiguously played out the speaker every 20 ms. This timing reconstruction is performed separately for each source of RTP packets in the conference. The sequence number can also be used by the receiver to estimate how many packets are being lost.

Since members of the working group join and leave during the conference, it is useful to know who is participating at any moment and how well they are receiving the audio data. For that purpose,

each instance of the audio application in the conference periodically multicasts a reception report plus the name of its user on the RTCP (control) port. The reception report indicates how well the current speaker is being received and may be used to control adaptive encodings. In addition to the user name, other identifying information may also be included subject to control bandwidth limits. A site sends the RTCP BYE packet (Section 6.5) when it leaves the conference.

2.2 Audio and Video Conference

If both audio and video media are used in a conference, they are transmitted as separate RTP sessions RTCP packets are transmitted for each medium using two different UDP port pairs and/or multicast addresses. There is no direct coupling at the RTP level between the audio and video sessions, except that a user participating in both sessions should use the same distinguished (canonical) name in the RTCP packets for both so that the sessions can be associated.

One motivation for this separation is to allow some participants in the conference to receive only one medium if they choose. Further explanation is given in Section 5.2. Despite the separation, synchronized playback of a source's audio and video can be achieved using timing information carried in the RTCP packets for both sessions.

2.3 Mixers and Translators

So far, we have assumed that all sites want to receive media data in the same format. However, this may not always be appropriate. Consider the case where participants in one area are connected through a low-speed link to the majority of the conference participants who enjoy high-speed network access. Instead of forcing everyone to use a lower-bandwidth, reduced-quality audio encoding, an RTP-level relay called a mixer may be placed near the low-bandwidth area. This mixer resynchronizes incoming audio packets to reconstruct the constant 20 ms spacing generated by the sender, mixes these reconstructed audio streams into a single stream, translates the audio encoding to a lower-bandwidth one and forwards the lower-bandwidth packet stream across the low-speed link. These packets might be unicast to a single recipient or multicast on a different address to multiple recipients. The RTP header includes a means for mixers to identify the sources that contributed to a mixed packet so that correct talker indication can be provided at the receivers.

Some of the intended participants in the audio conference may be connected with high bandwidth links but might not be directly reachable via IP multicast. For example, they might be behind an

application-level firewall that will not let any IP packets pass. For these sites, mixing may not be necessary, in which case another type of RTP-level relay called a translator may be used. Two translators are installed, one on either side of the firewall, with the outside one funneling all multicast packets received through a secure connection to the translator inside the firewall. The translator inside the firewall sends them again as multicast packets to a multicast group restricted to the site's internal network.

Mixers and translators may be designed for a variety of purposes. An example is a video mixer that scales the images of individual people in separate video streams and composites them into one video stream to simulate a group scene. Other examples of translation include the connection of a group of hosts speaking only IP/UDP to a group of hosts that understand only ST-II, or the packet-by-packet encoding translation of video streams from individual sources without resynchronization or mixing. Details of the operation of mixers and translators are given in Section 7.

3. Definitions

RTP payload: The data transported by RTP in a packet, for example audio samples or compressed video data. The payload format and interpretation are beyond the scope of this document.

RTP packet: A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources (see below), and the payload data. Some underlying protocols may require an encapsulation of the RTP packet to be defined. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation method (see Section 10).

RTCP packet: A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. The formats are defined in Section 6. Typically, multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol; this is enabled by the length field in the fixed header of each RTCP packet.

Port: The "abstraction that transport protocols use to distinguish among multiple destinations within a given host computer. TCP/IP protocols identify ports using small positive integers." [3] The transport selectors (TSEL) used by the OSI transport layer are equivalent to ports. RTP depends upon the lower-layer protocol to provide some mechanism such as ports to multiplex the RTP and RTCP packets of a session.

Transport address: The combination of a network address and port that identifies a transport-level endpoint, for example an IP address and a UDP port. Packets are transmitted from a source transport address to a destination transport address.

RTP session: The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses (one network address plus a port pair for RTP and RTCP). The destination transport address pair may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session, each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses.

Synchronization source (SSRC): The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. All packets from a synchronization source form part of the same timing and sequence number space, so a receiver groups packets by synchronization source for playback. Examples of synchronization sources include the sender of a stream of packets derived from a signal source such as a microphone or a camera, or an RTP mixer (see below). A synchronization source may change its data format, e.g., audio encoding, over time. The SSRC identifier is a randomly chosen value meant to be globally unique within a particular RTP session (see Section 8). A participant need not use the same SSRC identifier for all the RTP sessions in a multimedia session; the binding of the SSRC identifiers is provided through RTCP (see Section 6.4.1). If a participant generates multiple streams in one RTP session, for example from separate video cameras, each must be identified as a different SSRC.

Contributing source (CSRC): A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer (see below). The mixer inserts a list of the SSRC identifiers of the sources that contributed to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list. An example application is audio conferencing where a mixer indicates all the talkers whose speech was combined to produce the outgoing packet, allowing the receiver to indicate the current talker, even though all the audio packets contain the same SSRC identifier (that of the mixer).

End system: An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets. An end system can act as one or more synchronization sources in a particular RTP session, but typically only one.

Mixer: An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronized, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus, all data packets originating from a mixer will be identified as having the mixer as their synchronization source.

Translator: An intermediate system that forwards RTP packets with their synchronization source identifier intact. Examples of translators include devices that convert encodings without mixing, replicators from multicast to unicast, and application-level filters in firewalls.

Monitor: An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service for distribution monitoring, fault diagnosis and long-term statistics. The monitor function is likely to be built into the application(s) participating in the session, but may also be a separate application that does not otherwise participate and does not send or receive the RTP data packets. These are called third party monitors.

Non-RTP means: Protocols and mechanisms that may be needed in addition to RTP to provide a usable service. In particular, for multimedia conferences, a conference control application may distribute multicast addresses and keys for encryption, negotiate the encryption algorithm to be used, and define dynamic mappings between RTP payload type values and the payload formats they represent for formats that do not have a predefined payload type value. For simple applications, electronic mail or a conference database may also be used. The specification of such protocols and mechanisms is outside the scope of this document.

4. Byte Order, Alignment, and Time Format

All integer fields are carried in network byte order, that is, most significant byte (octet) first. This byte order is commonly known as big-endian. The transmission order is described in detail in [4]. Unless otherwise noted, numeric constants are in decimal (base 10).

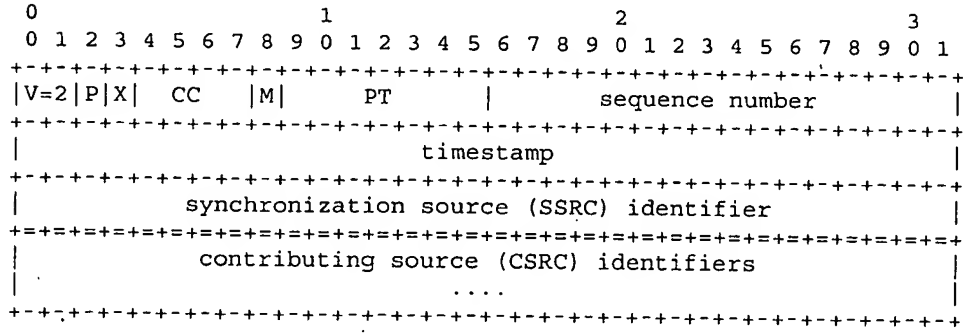
All header data is aligned to its natural length, i.e., 16-bit fields are aligned on even offsets, 32-bit fields are aligned at offsets divisible by four, etc. Octets designated as padding have the value zero.

Wallclock time (absolute time) is represented using the timestamp format of the Network Time Protocol (NTP), which is in seconds relative to 0h UTC on 1 January 1900 [5]. The full resolution NTP timestamp is a 64-bit unsigned fixed-point number with the integer part in the first 32 bits and the fractional part in the last 32 bits. In some fields where a more compact representation is appropriate, only the middle 32 bits are used; that is, the low 16 bits of the integer part and the high 16 bits of the fractional part. The high 16 bits of the integer part must be determined independently.

5. RTP Data Transfer Protocol

5.1 RTP Fixed Header Fields

The RTP header has the following format:



The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer. The fields have the following meaning:

version (V): 2 bits
: This field identifies the version of RTP. The version defined by this specification is two (2). (The value 1 is used by the first draft version of RTP and the value 0 is used by the protocol initially implemented in the "vat" audio tool.)

padding (P): 1 bit
: If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the

payload. The last octet of the padding contains a count of how many padding octets should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data unit.

extension (X): 1 bit
If the extension bit is set, the fixed header is followed by exactly one header extension, with a format defined in Section 5.3.1.

CSRC count (CC): 4 bits
The CSRC count contains the number of CSRC identifiers that follow the fixed header.

marker (M): 1 bit
The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream. A profile may define additional marker bits or specify that there is no marker bit by changing the number of bits in the payload type field (see Section 5.3).

payload type (PT): 7 bits
This field identifies the format of the RTP payload and determines its interpretation by the application. A profile specifies a default static mapping of payload type codes to payload formats. Additional payload type codes may be defined dynamically through non-RTP means (see Section 3). An initial set of default mappings for audio and video is specified in the companion profile Internet-Draft draft-ietf-avt-profile, and may be extended in future editions of the Assigned Numbers RFC [6]. An RTP sender emits a single RTP payload type at any given time; this field is not intended for multiplexing separate media streams (see Section 5.2).

sequence number: 16 bits
The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence. The initial value of the sequence number is random (unpredictable) to make known-plaintext attacks on encryption more difficult, even if the source itself does not encrypt, because the packets may flow through a translator that does. Techniques for choosing unpredictable numbers are discussed in [7].

timestamp: 32 bits
The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived

from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations (see Section 6.3.1). The resolution of the clock must be sufficient for the desired synchronization accuracy and for measuring packet arrival jitter (one tick per video frame is typically not sufficient). The clock frequency is dependent on the format of data carried as payload and is specified statically in the profile or payload format specification that defines the format, or may be specified dynamically for payload formats defined through non-RTP means. If RTP packets are generated periodically, the nominal sampling instant as determined from the sampling clock is to be used, not a reading of the system clock. As an example, for fixed-rate audio the timestamp clock would likely increment by one for each sampling period. If an audio application reads blocks covering 160 sampling periods from the input device, the timestamp would be increased by 160 for each such block, regardless of whether the block is transmitted in a packet or dropped as silent.

The initial value of the timestamp is random, as for the sequence number. Several consecutive RTP packets may have equal timestamps if they are (logically) generated at once, e.g., belong to the same video frame. Consecutive RTP packets may contain timestamps that are not monotonic if the data is not transmitted in the order it was sampled, as in the case of MPEG interpolated video frames. (The sequence numbers of the packets as transmitted will still be monotonic.)

SSRC: 32 bits
The SSRC field identifies the synchronization source. This identifier is chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC identifier. An example algorithm for generating a random identifier is presented in Appendix A.6. Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions. Section 8 describes the probability of collision along with a mechanism for resolving collisions and detecting RTP-level forwarding loops based on the uniqueness of the SSRC identifier. If a source changes its source transport address, it must also choose a new SSRC identifier to avoid being interpreted as a looped source.

CSRC list: 0 to 15 items, 32 bits each
The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are more than 15 contributing sources, only 15 may be identified. CSRC identifiers are

inserted by mixers, using the SSRC identifiers of contributing sources. For example, for audio packets the SSRC identifiers of all sources that were mixed together to create a packet are listed, allowing correct talker indication at the receiver.

5.2 Multiplexing RTP Sessions

For efficient protocol processing, the number of multiplexing points should be minimized, as described in the integrated layer processing design principle [1]. In RTP, multiplexing is provided by the destination transport address (network address and port number) which define an RTP session. For example, in a teleconference composed of audio and video media encoded separately, each medium should be carried in a separate RTP session with its own destination transport address. It is not intended that the audio and video be carried in a single RTP session and demultiplexed based on the payload type or SSRC fields. Interleaving packets with different payload types but using the same SSRC would introduce several problems:

1. If one payload type were switched during a session, there would be no general means to identify which of the old values the new one replaced.

2. An SSRC is defined to identify a single timing and sequence number space. Interleaving multiple payload types would require different timing spaces if the media clock rates differ and would require different sequence number spaces to tell which payload type suffered packet loss.

3. The RTCP sender and receiver reports (see Section 6.3) can only describe one timing and sequence number space per SSRC and do not carry a payload type field.

4. An RTP mixer would not be able to combine interleaved streams of incompatible media into one stream.

5. Carrying multiple media in one RTP session precludes: the use of different network paths or network resource allocations if appropriate; reception of a subset of the media if desired, for example just audio if video would exceed the available bandwidth; and receiver implementations that use separate processes for the different media, whereas using separate RTP sessions permits either single- or multiple-process implementations.

Using a different SSRC for each medium but sending them in the same RTP session would avoid the first three problems but not the last two.

5.3 Profile-Specific Modifications to the RTP Header

The existing RTP data packet header is believed to be complete for the set of functions required in common across all the application classes that RTP might support. However, in keeping with the ALF design principle, the header may be tailored through modifications or additions defined in a profile specification while still allowing profile-independent monitoring and recording tools to function.

- The marker bit and payload type field carry profile-specific information, but they are allocated in the fixed header since many applications are expected to need them and might otherwise have to add another 32-bit word just to hold them. The octet containing these fields may be redefined by a profile to suit different requirements, for example with a more or fewer marker bits. If there are any marker bits, one should be located in the most significant bit of the octet since profile-independent monitors may be able to observe a correlation between packet loss patterns and the marker bit.

- Additional information that is required for a particular payload format, such as a video encoding, should be carried in the payload section of the packet. This might be in a header that is always present at the start of the payload section, or might be indicated by a reserved value in the data pattern.

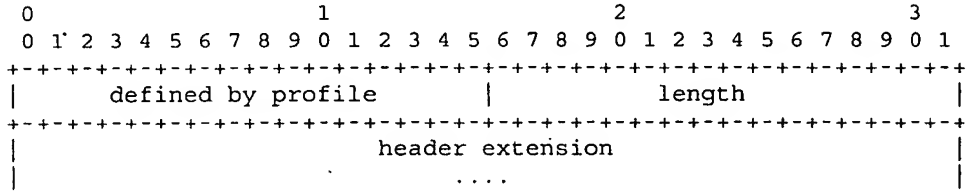
- If a particular class of applications needs additional functionality independent of payload format, the profile under which those applications operate should define additional fixed fields to follow immediately after the SSRC field of the existing fixed header. Those applications will be able to quickly and directly access the additional fields while profile-independent monitors or recorders can still process the RTP packets by interpreting only the first twelve octets.

If it turns out that additional functionality is needed in common across all profiles, then a new version of RTP should be defined to make a permanent change to the fixed header.

5.3.1 RTP Header Extension

An extension mechanism is provided to allow individual implementations to experiment with new payload-format-independent functions that require additional information to be carried in the RTP data packet header. This mechanism is designed so that the header extension may be ignored by other interoperating implementations that have not been extended.

Note that this header extension is intended only for limited use. Most potential uses of this mechanism would be better done another way, using the methods described in the previous section. For example, a profile-specific extension to the fixed header is less expensive to process because it is not conditional nor in a variable location. Additional information required for a particular payload format should not use this header extension, but should be carried in the payload section of the packet.



If the X bit in the RTP header is one, a variable-length header extension is appended to the RTP header, following the CSRC list if present. The header extension contains a 16-bit length field that counts the number of 32-bit words in the extension, excluding the four-octet extension header (therefore zero is a valid length). Only a single extension may be appended to the RTP data header. To allow multiple interoperating implementations to each experiment independently with different header extensions, or to allow a particular implementation to experiment with more than one type of header extension, the first 16 bits of the header extension are left open for distinguishing identifiers or parameters. The format of these 16 bits is to be defined by the profile specification under which the implementations are operating. This RTP specification does not define any header extensions itself.

6. RTP Control Protocol -- RTCP

The RTP control protocol (RTCP) is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. The underlying protocol must provide multiplexing of the data and control packets, for example using separate port numbers with UDP. RTCP performs four functions:

1. The primary function is to provide feedback on the quality of the data distribution. This is an integral part of the RTP's role as a transport protocol and is related to the flow and congestion control functions of other transport protocols. The feedback may be directly useful for control of adaptive encodings [8,9], but experiments with IP

multicasting have shown that it is also critical to get feedback from the receivers to diagnose faults in the distribution. Sending reception feedback reports to all participants allows one who is observing problems to evaluate whether those problems are local or global. With a distribution mechanism like IP multicast, it is also possible for an entity such as a network service provider who is not otherwise involved in the session to receive the feedback information and act as a third-party monitor to diagnose network problems. This feedback function is performed by the RTCP sender and receiver reports, described below in Section 6.3.

2. RTCP carries a persistent transport-level identifier for an RTP source called the canonical name or CNAME, Section 6.4.1. Since the SSRC identifier may change if a conflict is discovered or a program is restarted, receivers require the CNAME to keep track of each participant. Receivers also require the CNAME to associate multiple data streams from a given participant in a set of related RTP sessions, for example to synchronize audio and video.

3. The first two functions require that all participants send RTCP packets, therefore the rate must be controlled in order for RTP to scale up to a large number of participants. By having each participant send its control packets to all the others, each can independently observe the number of participants. This number is used to calculate the rate at which the packets are sent, as explained in Section 6.2..

4. A fourth, optional function is to convey minimal session control information, for example participant identification to be displayed in the user interface. This is most likely to be useful in "loosely controlled" sessions where participants enter and leave without membership control or parameter negotiation. RTCP serves as a convenient channel to reach all the participants, but it is not necessarily expected to support all the control communication requirements of an application. A higher-level session control protocol, which is beyond the scope of this document, may be needed.

Functions 1-3 are mandatory when RTP is used in the IP multicast environment, and are recommended for all environments. RTP application designers are advised to avoid mechanisms that can only work in unicast mode and will not scale to larger numbers.

6.1 RTCP Packet Format

This specification defines several RTCP packet types to carry a variety of control information:

SR: Sender report, for transmission and reception statistics from participants that are active senders

RR: Receiver report, for reception statistics from participants that are not active senders

SDES: Source description items, including CNAME

BYE: Indicates end of participation

APP: Application specific functions

Each RTCP packet begins with a fixed part similar to that of RTP data packets, followed by structured elements that may be of variable length according to the packet type but always end on a 32-bit boundary. The alignment requirement and a length field in the fixed part are included to make RTCP packets "stackable". Multiple RTCP packets may be concatenated without any intervening separators to form a compound RTCP packet that is sent in a single packet of the lower layer protocol, for example UDP. There is no explicit count of individual RTCP packets in the compound packet since the lower layer protocols are expected to provide an overall length to determine the end of the compound packet.

Each individual RTCP packet in the compound packet may be processed independently with no requirements upon the order or combination of packets. However, in order to perform the functions of the protocol, the following constraints are imposed:

- Reception statistics (in SR or RR) should be sent as often as bandwidth constraints will allow to maximize the resolution of the statistics, therefore each periodically transmitted compound RTCP packet should include a report packet.

- New receivers need to receive the CNAME for a source as soon as possible to identify the source and to begin associating media for purposes such as lip-sync, so each compound RTCP packet should also include the SDES CNAME.

- The number of packet types that may appear first in the compound packet should be limited to increase the number of constant bits in the first word and the probability of successfully validating RTCP packets against misaddressed RTP

data packets or other unrelated packets.

Thus, all RTCP packets must be sent in a compound packet of at least two individual packets, with the following format recommended:

Encryption prefix: If and only if the compound packet is to be encrypted, it is prefixed by a random 32-bit quantity redrawn for every compound packet transmitted.

SR or RR: The first RTCP packet in the compound packet must always be a report packet to facilitate header validation as described in Appendix A.2. This is true even if no data has been sent nor received, in which case an empty RR is sent, and even if the only other RTCP packet in the compound packet is a BYE.

Additional RRs: If the number of sources for which reception statistics are being reported exceeds 31, the number that will fit into one SR or RR packet, then additional RR packets should follow the initial report packet.

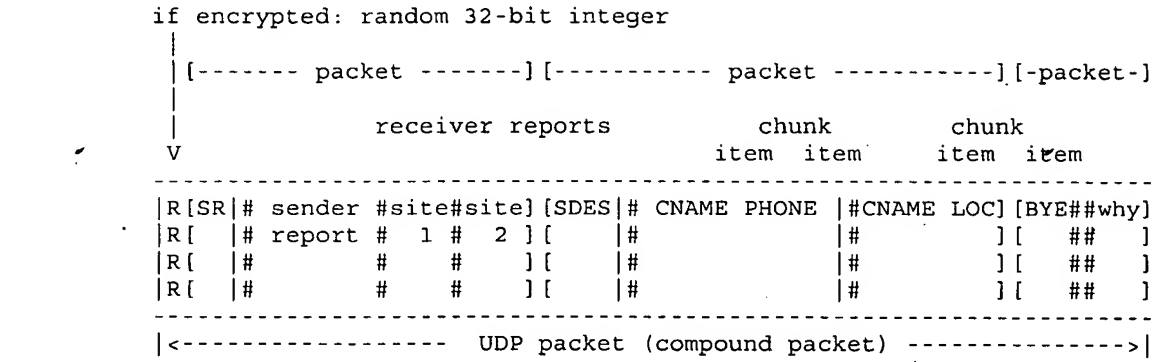
SDES: An SDES packet containing a CNAME item must be included in each compound RTCP packet. Other source description items may optionally be included if required by a particular application, subject to bandwidth constraints (see Section 6.2.2).

BYE or APP: Other RTCP packet types, including those yet to be defined, may follow in any order, except that BYE should be the last packet sent with a given SSRC/CSRC. Packet types may appear more than once.

It is advisable for translators and mixers to combine individual RTCP packets from the multiple sources they are forwarding into one compound packet whenever feasible in order to amortize the packet overhead (see Section 7). An example RTCP compound packet as might be produced by a mixer is shown in Fig. 1. If the overall length of a compound packet would exceed the maximum transmission unit (MTU) of the network path, it may be segmented into multiple shorter compound packets to be transmitted in separate packets of the underlying protocol. Note that each of the compound packets must begin with an SR or RR packet.

An implementation may ignore incoming RTCP packets with types unknown to it. Additional RTCP packet types may be registered with the Internet Assigned Numbers Authority (IANA).

6.2 RTCP Transmission Interval



#: SSRC/CSRC

Figure 1: Example of an RTCP compound packet

RTP is designed to allow an application to scale automatically over session sizes ranging from a few participants to thousands. For example, in an audio conference the data traffic is inherently self-limiting because only one or two people will speak at a time, so with multicast distribution the data rate on any given link remains relatively constant independent of the number of participants. However, the control traffic is not self-limiting. If the reception reports from each participant were sent at a constant rate, the control traffic would grow linearly with the number of participants. Therefore, the rate must be scaled down.

For each session, it is assumed that the data traffic is subject to an aggregate limit called the "session bandwidth" to be divided among the participants. This bandwidth might be reserved and the limit enforced by the network, or it might just be a reasonable share. The session bandwidth may be chosen based or some cost or a priori knowledge of the available network bandwidth for the session. It is somewhat independent of the media encoding, but the encoding choice may be limited by the session bandwidth. The session bandwidth parameter is expected to be supplied by a session management application when it invokes a media application, but media applications may also set a default based on the single-sender data bandwidth for the encoding selected for the session. The application may also enforce bandwidth limits based on multicast scope rules or other criteria.

Bandwidth calculations for control and data traffic include lower-layer transport and network protocols (e.g., UDP and IP) since that is what the resource reservation system would need to know. The application can also be expected to know which of these protocols are in use. Link level headers are not included in the calculation since the packet will be encapsulated with different link level headers as it travels.

The control traffic should be limited to a small and known fraction of the session bandwidth: small so that the primary function of the transport protocol to carry data is not impaired; known so that the control traffic can be included in the bandwidth specification given to a resource reservation protocol, and so that each participant can independently calculate its share. It is suggested that the fraction of the session bandwidth allocated to RTCP be fixed at 5%. While the value of this and other constants in the interval calculation is not critical, all participants in the session must use the same values so the same interval will be calculated. Therefore, these constants should be fixed for a particular profile.

The algorithm described in Appendix A.7 was designed to meet the goals outlined above. It calculates the interval between sending compound RTCP packets to divide the allowed control traffic bandwidth among the participants. This allows an application to provide fast response for small sessions where, for example, identification of all participants is important, yet automatically adapt to large sessions. The algorithm incorporates the following characteristics:

- Senders are collectively allocated at least 1/4 of the control traffic bandwidth so that in sessions with a large number of receivers but a small number of senders, newly joining participants will more quickly receive the CNAME for the sending sites.

- The calculated interval between RTCP packets is required to be greater than a minimum of 5 seconds to avoid having bursts of RTCP packets exceed the allowed bandwidth when the number of participants is small and the traffic isn't smoothed according to the law of large numbers.

- The interval between RTCP packets is varied randomly over the range [0.5,1.5] times the calculated interval to avoid unintended synchronization of all participants [10]. The first RTCP packet sent after joining a session is also delayed by a random variation of half the minimum RTCP interval in case the application is started at multiple sites simultaneously, for example as initiated by a session announcement.

- o A dynamic estimate of the average compound RTCP packet size is calculated, including all those received and sent, to automatically adapt to changes in the amount of control information carried.

This algorithm may be used for sessions in which all participants are allowed to send. In that case, the session bandwidth parameter is the product of the individual sender's bandwidth times the number of participants, and the RTCP bandwidth is 5% of that.

6.2.1 Maintaining the number of session members

Calculation of the RTCP packet interval depends upon an estimate of the number of sites participating in the session. New sites are added to the count when they are heard, and an entry for each is created in a table indexed by the SSRC or CSRC identifier (see Section 8.2) to keep track of them. New entries may not be considered valid until multiple packets carrying the new SSRC have been received (see Appendix A.1). Entries may be deleted from the table when an RTCP BYE packet with the corresponding SSRC identifier is received.

A participant may mark another site inactive, or delete it if not yet valid, if no RTP or RTCP packet has been received for a small number of RTCP report intervals (5 is suggested). This provides some robustness against packet loss. All sites must calculate roughly the same value for the RTCP report interval in order for this timeout to work properly.

Once a site has been validated, then if it is later marked inactive the state for that site should still be retained and the site should continue to be counted in the total number of sites sharing RTCP bandwidth for a period long enough to span typical network partitions. This is to avoid excessive traffic, when the partition heals, due to an RTCP report interval that is too small. A timeout of 30 minutes is suggested. Note that this is still larger than 5 times the largest value to which the RTCP report interval is expected to usefully scale, about 2 to 5 minutes.

6.2.2 Allocation of source description bandwidth

This specification defines several source description (SDES) items in addition to the mandatory CNAME item, such as NAME (personal name) and EMAIL (email address). It also provides a means to define new application-specific RTCP packet types. Applications should exercise caution in allocating control bandwidth to this additional information because it will slow down the rate at which reception reports and CNAME are sent, thus impairing the performance of the protocol. It is recommended that no more than 20% of the RTCP

bandwidth allocated to a single participant be used to carry the additional information. Furthermore, it is not intended that all SDES items should be included in every application. Those that are included should be assigned a fraction of the bandwidth according to their utility. Rather than estimate these fractions dynamically, it is recommended that the percentages be translated statically into report interval counts based on the typical length of an item.

For example, an application may be designed to send only CNAME, NAME and EMAIL and not any others. NAME might be given much higher priority than EMAIL because the NAME would be displayed continuously in the application's user interface, whereas EMAIL would be displayed only when requested. At every RTCP interval, an RR packet and an SDES packet with the CNAME item would be sent. For a small session operating at the minimum interval, that would be every 5 seconds on the average. Every third interval (15 seconds), one extra item would be included in the SDES packet. Seven out of eight times this would be the NAME item, and every eighth time (2 minutes) it would be the EMAIL item.

When multiple applications operate in concert using cross-application binding through a common CNAME for each participant, for example in a multimedia conference composed of an RTP session for each medium, the additional SDES information might be sent in only one RTP session. The other sessions would carry only the CNAME item.

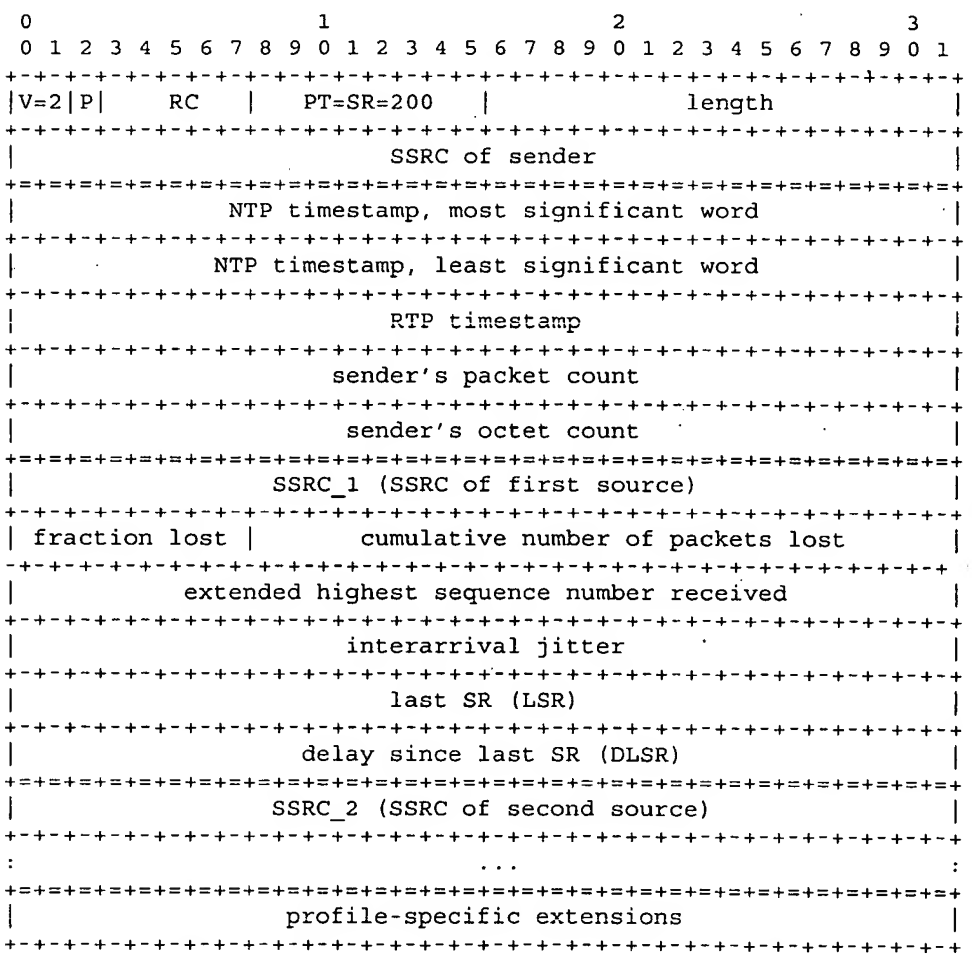
6.3 Sender and Receiver Reports

RTP receivers provide reception quality feedback using RTCP report packets which may take one of two forms depending upon whether or not the receiver is also a sender. The only difference between the sender report (SR) and receiver report (RR) forms, besides the packet type code, is that the sender report includes a 20-byte sender information section for use by active senders. The SR is issued if a site has sent any data packets during the interval since issuing the last report or the previous one, otherwise the RR is issued.

Both the SR and RR forms include zero or more reception report blocks, one for each of the synchronization sources from which this receiver has received RTP data packets since the last report. Reports are not issued for contributing sources listed in the CSRC list. Each reception report block provides statistics about the data received from the particular source indicated in that block. Since a maximum of 31 reception report blocks will fit in an SR or RR packet, additional RR packets may be stacked after the initial SR or RR packet as needed to contain the reception reports for all sources heard during the interval since the last report.

The next sections define the formats of the two reports, how they may be extended in a profile-specific manner if an application requires additional feedback information, and how the reports may be used. Details of reception reporting by translators and mixers is given in Section 7.

6.3.1 SR: Sender report RTCP packet



The sender report packet consists of three sections, possibly followed by a fourth profile-specific extension section if defined. The first section, the header, is 8 octets long. The fields have the following meaning:

version (V): 2 bits
Identifies the version of RTP, which is the same in RTCP packets as in RTP data packets. The version defined by this specification is two (2).

padding (P): 1 bit
If the padding bit is set, this RTCP packet contains some additional padding octets at the end which are not part of the control information. The last octet of the padding is a count of how many padding octets should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes. In a compound RTCP packet, padding should only be required on the last individual packet because the compound packet is encrypted as a whole.

reception report count (RC): 5 bits
The number of reception report blocks contained in this packet. A value of zero is valid.

packet type (PT): 8 bits
Contains the constant 200 to identify this as an RTCP SR packet.

length: 16 bits
The length of this RTCP packet in 32-bit words minus one, including the header and any padding. (The offset of one makes zero a valid length and avoids a possible infinite loop in scanning a compound RTCP packet, while counting 32-bit words avoids a validity check for a multiple of 4.)

SSRC: 32 bits
The synchronization source identifier for the originator of this SR packet.

The second section, the sender information, is 20 octets long and is present in every sender report packet. It summarizes the data transmissions from this sender. The fields have the following meaning:

NTP timestamp: 64 bits
Indicates the wallclock time when this report was sent so that it may be used in combination with timestamps returned in reception reports from other receivers to measure round-trip propagation to those receivers. Receivers should expect that the measurement accuracy of the timestamp may be limited to far less than the resolution of the NTP timestamp. The measurement uncertainty of the timestamp is not indicated as it may not be known. A sender that can keep track of elapsed time but has no notion of wallclock time may use the elapsed time since joining

the session instead. This is assumed to be less than 68 years, so the high bit will be zero. It is permissible to use the sampling clock to estimate elapsed wallclock time. A sender that has no notion of wallclock or elapsed time may set the NTP timestamp to zero.

RTP timestamp: 32 bits
Corresponds to the same time as the NTP timestamp (above), but in the same units and with the same random offset as the RTP timestamps in data packets. This correspondence may be used for intra- and inter-media synchronization for sources whose NTP timestamps are synchronized, and may be used by media-independent receivers to estimate the nominal RTP clock frequency. Note that in most cases this timestamp will not be equal to the RTP timestamp in any adjacent data packet. Rather, it is calculated from the corresponding NTP timestamp using the relationship between the RTP timestamp counter and real time as maintained by periodically checking the wallclock time at a sampling instant.

sender's packet count: 32 bits
The total number of RTP data packets transmitted by the sender since starting transmission up until the time this SR packet was generated. The count is reset if the sender changes its SSRC identifier.

sender's octet count: 32 bits
The total number of payload octets (i.e., not including header or padding) transmitted in RTP data packets by the sender since starting transmission up until the time this SR packet was generated. The count is reset if the sender changes its SSRC identifier. This field can be used to estimate the average payload data rate.

The third section contains zero or more reception report blocks depending on the number of other sources heard by this sender since the last report. Each reception report block conveys statistics on the reception of RTP packets from a single synchronization source. Receivers do not carry over statistics when a source changes its SSRC identifier due to a collision. These statistics are:

SSRC_n (source identifier): 32 bits
The SSRC identifier of the source to which the information in this reception report block pertains.

fraction lost: 8 bits
The fraction of RTP data packets from source SSRC_n lost since the previous SR or RR packet was sent, expressed as a fixed

point number with the binary point at the left edge of the field. (That is equivalent to taking the integer part after multiplying the loss fraction by 256.) This fraction is defined to be the number of packets lost divided by the number of packets expected, as defined in the next paragraph. An implementation is shown in Appendix A.3. If the loss is negative due to duplicates, the fraction lost is set to zero. Note that a receiver cannot tell whether any packets were lost after the last one received, and that there will be no reception report block issued for a source if all packets from that source sent during the last reporting interval have been lost.

cumulative number of packets lost: 24 bits
The total number of RTP data packets from source SSRC_n that have been lost since the beginning of reception. This number is defined to be the number of packets expected less the number of packets actually received, where the number of packets received includes any which are late or duplicates. Thus packets that arrive late are not counted as lost, and the loss may be negative if there are duplicates. The number of packets expected is defined to be the extended last sequence number received, as defined next, less the initial sequence number received. This may be calculated as shown in Appendix A.3.

extended highest sequence number received: 32 bits
The low 16 bits contain the highest sequence number received in an RTP data packet from source SSRC_n, and the most significant 16 bits extend that sequence number with the corresponding count of sequence number cycles, which may be maintained according to the algorithm in Appendix A.1. Note that different receivers within the same session will generate different extensions to the sequence number if their start times differ significantly.

interarrival jitter: 32 bits
An estimate of the statistical variance of the RTP data packet interarrival time, measured in timestamp units and expressed as an unsigned integer. The interarrival jitter J is defined to be the mean deviation (smoothed absolute value) of the difference D in packet spacing at the receiver compared to the sender for a pair of packets. As shown in the equation below, this is equivalent to the difference in the "relative transit time" for the two packets; the relative transit time is the difference between a packet's RTP timestamp and the receiver's clock at the time of arrival, measured in the same units.

If Si is the RTP timestamp from packet i, and Ri is the time of arrival in RTP timestamp units for packet i, then for two packets i and j, D may be expressed as

$$D(i,j)=(Rj-Ri)-(Sj-Si)=(Rj-Sj)-(Ri-Si)$$

The interarrival jitter is calculated continuously as each data packet i is received from source SSRC_n, using this difference D for that packet and the previous packet i-1 in order of arrival (not necessarily in sequence), according to the formula

$$J=J+(|D(i-1,i)|-J)/16$$

Whenever a reception report is issued, the current value of J is sampled.

The jitter calculation is prescribed here to allow profile-independent monitors to make valid interpretations of reports coming from different implementations. This algorithm is the optimal first-order estimator and the gain parameter 1/16 gives a good noise reduction ratio while maintaining a reasonable rate of convergence [11]. A sample implementation is shown in Appendix A.8.

last SR timestamp (LSR): 32 bits
: The middle 32 bits out of 64 in the NTP timestamp (as explained in Section 4) received as part of the most recent RTCP sender report (SR) packet from source SSRC_n. If no SR has been received yet, the field is set to zero.

delay since last SR (DLSR): 32 bits
: The delay, expressed in units of 1/65536 seconds, between receiving the last SR packet from source SSRC_n and sending this reception report block. If no SR packet has been received yet from SSRC_n, the DLSR field is set to zero.

Let SSRC_r denote the receiver issuing this receiver report. Source SSRC_n can compute the round propagation delay to SSRC_r by recording the time A when this reception report block is received. It calculates the total round-trip time A-LSR using the last SR timestamp (LSR) field, and then subtracting this field to leave the round-trip propagation delay as (A- LSR - DLSR). This is illustrated in Fig. 2.

This may be used as an approximate measure of distance to cluster receivers, although some links have very asymmetric delays.

6.3.2 RR: Receiver report RTCP packet

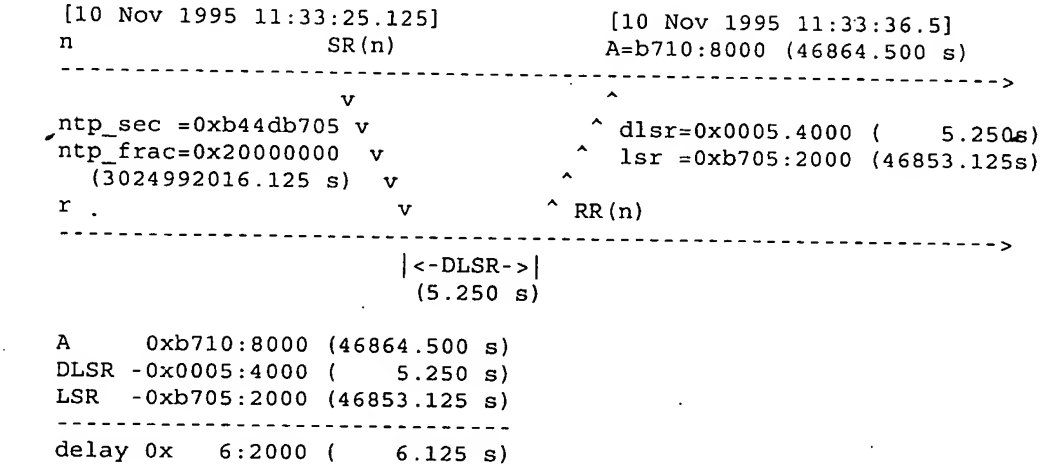
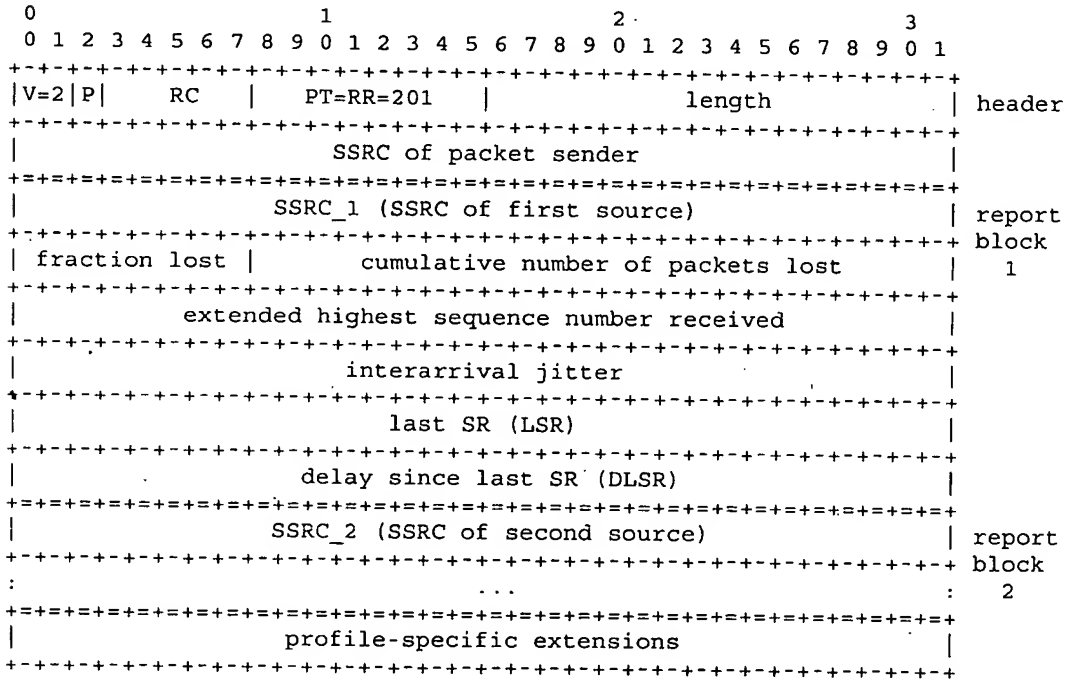


Figure 2: Example for round-trip time computation



The format of the receiver report (RR) packet is the same as that of the SR packet except that the packet type field contains the constant 201 and the five words of sender information are omitted (these are the NTP and RTP timestamps and sender's packet and octet counts). The remaining fields have the same meaning as for the SR packet.

An empty RR packet (RC = 0) is put at the head of a compound RTCP packet when there is no data transmission or reception to report.

6.3.3 Extending the sender and receiver reports

A profile should define profile- or application-specific extensions to the sender report and receiver if there is additional information that should be reported regularly about the sender or receivers. This method should be used in preference to defining another RTCP packet type because it requires less overhead:

- fewer octets in the packet (no RTCP header or SSRC field);
- simpler and faster parsing because applications running under that profile would be programmed to always expect the extension fields in the directly accessible location after the reception reports.

If additional sender information is required, it should be included first in the extension for sender reports, but would not be present in receiver reports. If information about receivers is to be included, that data may be structured as an array of blocks parallel to the existing array of reception report blocks; that is, the number of blocks would be indicated by the RC field.

6.3.4 Analyzing sender and receiver reports

It is expected that reception quality feedback will be useful not only for the sender but also for other receivers and third-party monitors. The sender may modify its transmissions based on the feedback; receivers can determine whether problems are local, regional or global; network managers may use profile-independent monitors that receive only the RTCP packets and not the corresponding RTP data packets to evaluate the performance of their networks for multicast distribution.

Cumulative counts are used in both the sender information and receiver report blocks so that differences may be calculated between any two reports to make measurements over both short and long time periods, and to provide resilience against the loss of a report. The difference between the last two reports received can be used to estimate the recent quality of the distribution. The NTP timestamp is

included so that rates may be calculated from these differences over the interval between two reports. Since that timestamp is independent of the clock rate for the data encoding, it is possible to implement encoding- and profile-independent quality monitors.

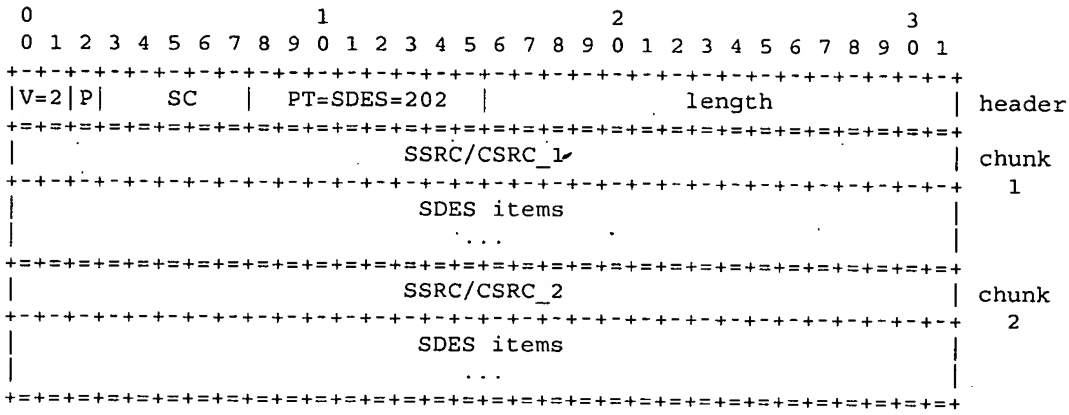
An example calculation is the packet loss rate over the interval between two reception reports. The difference in the cumulative number of packets lost gives the number lost during that interval. The difference in the extended last sequence numbers received gives the number of packets expected during the interval. The ratio of these two is the packet loss fraction over the interval. This ratio should equal the fraction lost field if the two reports are consecutive, but otherwise not. The loss rate per second can be obtained by dividing the loss fraction by the difference in NTP timestamps, expressed in seconds. The number of packets received is the number of packets expected minus the number lost. The number of packets expected may also be used to judge the statistical validity of any loss estimates. For example, 1 out of 5 packets lost has a lower significance than 200 out of 1000.

From the sender information, a third-party monitor can calculate the average payload data rate and the average packet rate over an interval without receiving the data. Taking the ratio of the two gives the average payload size. If it can be assumed that packet loss is independent of packet size, then the number of packets received by a particular receiver times the average payload size (or the corresponding packet size) gives the apparent throughput available to that receiver.

In addition to the cumulative counts which allow long-term packet loss measurements using differences between reports, the fraction lost field provides a short-term measurement from a single report. This becomes more important as the size of a session scales up enough that reception state information might not be kept for all receivers or the interval between reports becomes long enough that only one report might have been received from a particular receiver.

The interarrival jitter field provides a second short-term measure of network congestion. Packet loss tracks persistent congestion while the jitter measure tracks transient congestion. The jitter measure may indicate congestion before it leads to packet loss. Since the interarrival jitter field is only a snapshot of the jitter at the time of a report, it may be necessary to analyze a number of reports from one receiver over time or from multiple receivers, e.g., within a single network.

6.4 SDES: Source description RTCP packet



The SDES packet is a three-level structure composed of a header and zero or more chunks, each of of which is composed of items describing the source identified in that chunk. The items are described individually in subsequent sections.

version (V), padding (P), length:
As described for the SR packet (see Section 6.3.1).

packet type (PT): 8 bits
Contains the constant 202 to identify this as an RTCP SDES packet.

source count (SC): 5 bits
The number of SSRC/CSRC chunks contained in this SDES packet. A value of zero is valid but useless.

Each chunk consists of an SSRC/CSRC identifier followed by a list of zero or more items, which carry information about the SSRC/CSRC. Each chunk starts on a 32-bit boundary. Each item consists of an 8-bit type field, an 8-bit octet count describing the length of the text (thus, not including this two-octet header), and the text itself. Note that the text can be no longer than 255 octets, but this is consistent with the need to limit RTCP bandwidth consumption.

The text is encoded according to the UTF-2 encoding specified in Annex F of ISO standard 10646 [12,13]. This encoding is also known as UTF-8 or UTF-FSS. It is described in "File System Safe UCS Transformation Format (FSS_UTF)", X/Open Preliminary Specification, Document Number P316 and Unicode Technical Report #4. US-ASCII is a subset of this encoding and requires no additional encoding. The

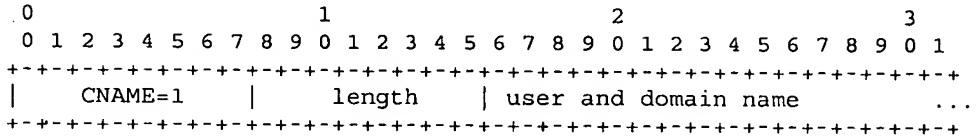
presence of multi-octet encodings is indicated by setting the most significant bit of a character to a value of one.

Items are contiguous, i.e., items are not individually padded to a 32-bit boundary. Text is not null terminated because some multi-octet encodings include null octets. The list of items in each chunk is terminated by one or more null octets, the first of which is interpreted as an item type of zero to denote the end of the list, and the remainder as needed to pad until the next 32-bit boundary. A chunk with zero items (four null octets) is valid but useless.

End systems send one SDES packet containing their own source identifier (the same as the SSRC in the fixed RTP header). A mixer sends one SDES packet containing a chunk for each contributing source from which it is receiving SDES information, or multiple complete SDES packets in the format above if there are more than 31 such sources (see Section 7).

The SDES items currently defined are described in the next sections. Only the CNAME item is mandatory. Some items shown here may be useful only for particular profiles, but the item types are all assigned from one common space to promote shared use and to simplify profile-independent applications. Additional items may be defined in a profile by registering the type numbers with IANA.

6.4.1 CNAME: Canonical end-point identifier SDES item



The CNAME identifier has the following properties:

- Because the randomly allocated SSRC identifier may change if a conflict is discovered or if a program is restarted, the CNAME item is required to provide the binding from the SSRC identifier to an identifier for the source that remains constant.

- Like the SSRC identifier, the CNAME identifier should also be unique among all participants within one RTP session.

- To provide a binding across multiple media tools used by one participant in a set of related RTP sessions, the CNAME should be fixed for that participant.

- To facilitate third-party monitoring, the CNAME should be suitable for either a program or a person to locate the source.

Therefore, the CNAME should be derived algorithmically and not entered manually, when possible. To meet these requirements, the following format should be used unless a profile specifies an alternate syntax or semantics. The CNAME item should have the format "user@host", or "host" if a user name is not available as on single-user systems. For both formats, "host" is either the fully qualified domain name of the host from which the real-time data originates, formatted according to the rules specified in RFC 1034 [14], RFC 1035 [15] and Section 2.1 of RFC 1123 [16]; or the standard ASCII representation of the host's numeric address on the interface used for the RTP communication. For example, the standard ASCII representation of an IP Version 4 address is "dotted decimal", also known as dotted quad. Other address types are expected to have ASCII representations that are mutually unique. The fully qualified domain name is more convenient for a human observer and may avoid the need to send a NAME item in addition, but it may be difficult or impossible to obtain reliably in some operating environments. Applications that may be run in such environments should use the ASCII representation of the address instead.

Examples are "doe@sleepy.megacorp.com" or "doe@192.0.2.89" for a multi-user system. On a system with no user name, examples would be "sleepy.megacorp.com" or "192.0.2.89".

The user name should be in a form that a program such as "finger" or "talk" could use, i.e., it typically is the login name rather than the personal name. The host name is not necessarily identical to the one in the participant's electronic mail address.

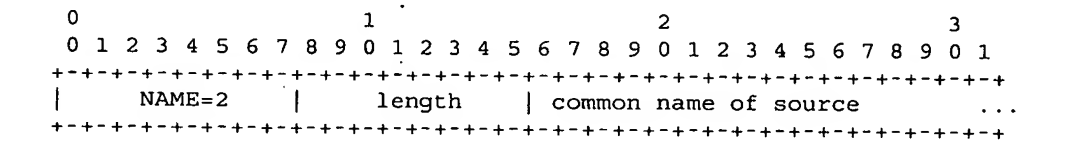
This syntax will not provide unique identifiers for each source if an application permits a user to generate multiple sources from one host. Such an application would have to rely on the SSRC to further identify the source, or the profile for that application would have to specify additional syntax for the CNAME identifier.

If each application creates its CNAME independently, the resulting CNAMEs may not be identical as would be required to provide a binding across multiple media tools belonging to one participant in a set of related RTP sessions. If cross-media binding is required, it may be necessary for the CNAME of each tool to be externally configured with the same value by a coordination tool.

Application writers should be aware that private network address assignments such as the Net-10 assignment proposed in RFC 1597 [17] may create network addresses that are not globally unique. This would

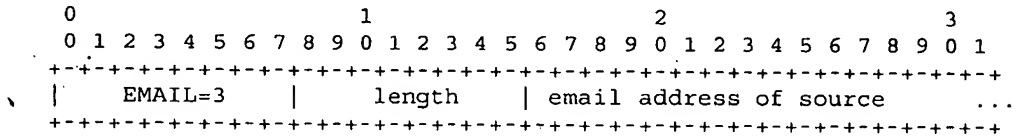
lead to non-unique CNAMEs if hosts with private addresses and no direct IP connectivity to the public Internet have their RTP packets forwarded to the public Internet through an RTP-level translator. (See also RFC 1627 [18].) To handle this case, applications may provide a means to configure a unique CNAME, but the burden is on the translator to translate CNAMEs from private addresses to public addresses if necessary to keep private addresses from being exposed.

6.4.2 NAME: User name SDES item



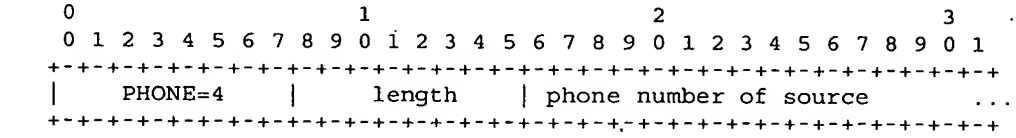
This is the real name used to describe the source, e.g., "John Doe, Bit Recycler, Megacorp". It may be in any form desired by the user. For applications such as conferencing, this form of name may be the most desirable for display in participant lists, and therefore might be sent most frequently of those items other than CNAME. Profiles may establish such priorities. The NAME value is expected to remain constant at least for the duration of a session. It should not be relied upon to be unique among all participants in the session.

6.4.3 EMAIL: Electronic mail address SDES item



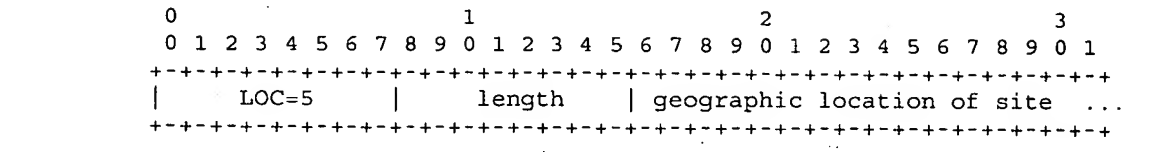
The email address is formatted according to RFC 822 [19], for example, "John.Doe@megacorp.com". The EMAIL value is expected to remain constant for the duration of a session.

6.4.4 PHONE: Phone number SDES item



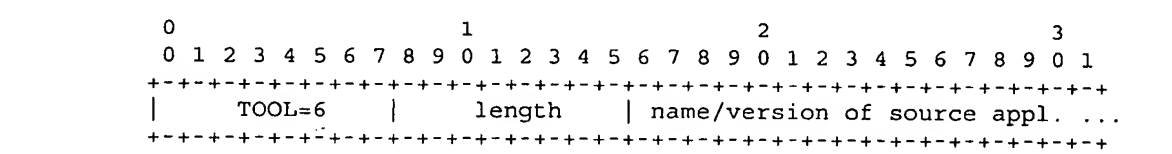
The phone number should be formatted with the plus sign replacing the international access code. For example, "+1 908 555 1212" for a number in the United States.

6.4.5 LOC: Geographic user location SDES item



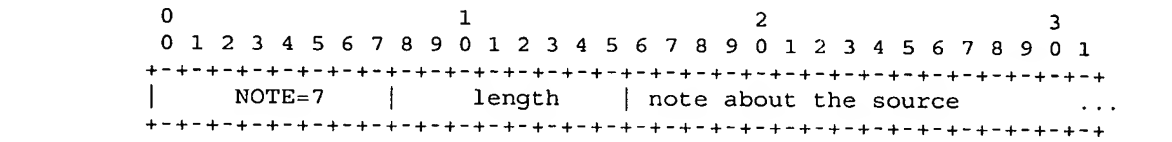
Depending on the application, different degrees of detail are appropriate for this item. For conference applications, a string like "Murray Hill, New Jersey" may be sufficient, while, for an active badge system, strings like "Room 2A244, AT&T BL MH" might be appropriate. The degree of detail is left to the implementation and/or user, but format and content may be prescribed by a profile. The LOC value is expected to remain constant for the duration of a session, except for mobile hosts.

6.4.6 TOOL: Application or tool name SDES item



A string giving the name and possibly version of the application generating the stream, e.g., "videotool 1.2". This information may be useful for debugging purposes and is similar to the Mailer or Mail-System-Version SMTP headers. The TOOL value is expected to remain constant for the duration of the session.

6.4.7 NOTE: Notice/status SDES item

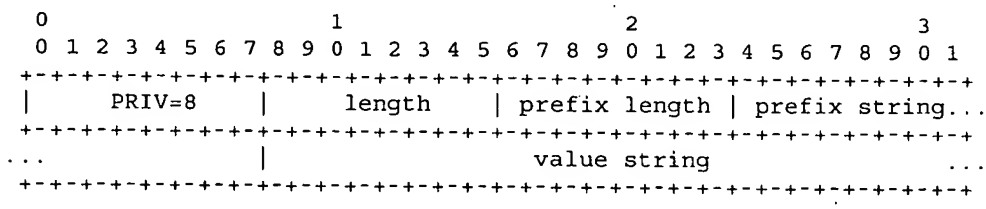


The following semantics are suggested for this item, but these or other semantics may be explicitly defined by a profile. The NOTE item is intended for transient messages describing the current state of the source, e.g., "on the phone, can't talk". Or, during a seminar, this item might be used to convey the title of the talk. It should be used only to carry exceptional information and should not be included routinely by all participants because this would slow down the rate at which reception reports and CNAME are sent, thus impairing the performance of the protocol. In particular, it should not be included

as an item in a user's configuration file nor automatically generated as in a quote-of-the-day.

Since the NOTE item may be important to display while it is active, the rate at which other non-CNAME items such as NAME are transmitted might be reduced so that the NOTE item can take that part of the RTCP bandwidth. When the transient message becomes inactive, the NOTE item should continue to be transmitted a few times at the same repetition rate but with a string of length zero to signal the receivers. However, receivers should also consider the NOTE item inactive if it is not received for a small multiple of the repetition rate, or perhaps 20-30 RTCP intervals.

6.4.8 PRIV: Private extensions SDES item

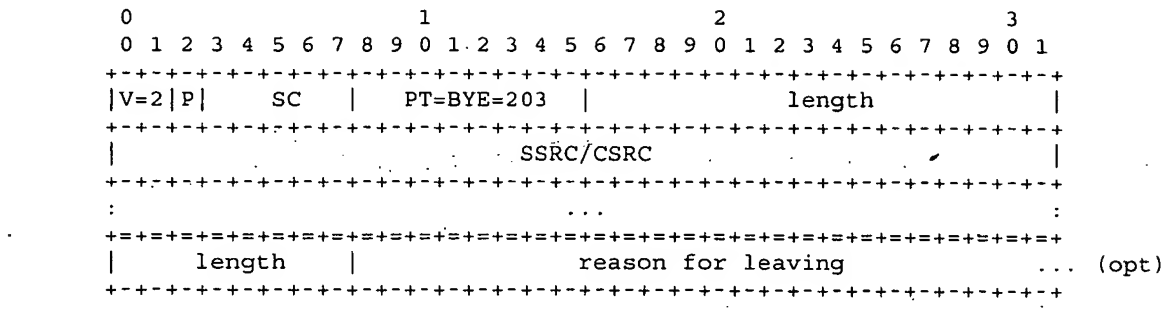


This item is used to define experimental or application-specific SDES extensions. The item contains a prefix consisting of a length-string pair, followed by the value string filling the remainder of the item and carrying the desired information. The prefix length field is 8 bits long. The prefix string is a name chosen by the person defining the PRIV item to be unique with respect to other PRIV items this application might receive. The application creator might choose to use the application name plus an additional subtype identification if needed. Alternatively, it is recommended that others choose a name based on the entity they represent, then coordinate the use of the name within that entity.

Note that the prefix consumes some space within the item's total length of 255 octets, so the prefix should be kept as short as possible. This facility and the constrained RTCP bandwidth should not be overloaded; it is not intended to satisfy all the control communication requirements of all applications.

SDES PRIV prefixes will not be registered by IANA. If some form of the PRIV item proves to be of general utility, it should instead be assigned a regular SDES item type registered with IANA so that no prefix is required. This simplifies use and increases transmission efficiency.

6.5 BYE: Goodbye RTCP packet



The BYE packet indicates that one or more sources are no longer active.

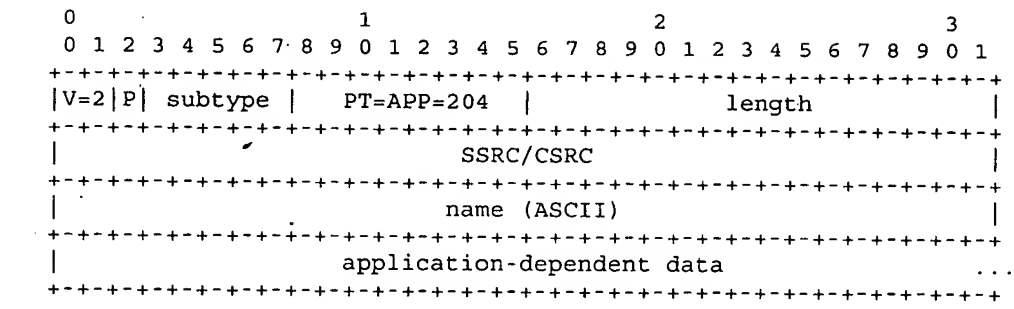
version (V), padding (P), length:
As described for the SR packet (see Section 6.3.1).

packet type (PT): 8 bits
Contains the constant 203 to identify this as an RTCP BYE packet.

source count (SC): 5 bits
The number of SSRC/CSRC identifiers included in this BYE packet. A count value of zero is valid, but useless.

If a BYE packet is received by a mixer, the mixer forwards the BYE packet with the SSRC/CSRC identifier(s) unchanged. If a mixer shuts down, it should send a BYE packet listing all contributing sources it handles, as well as its own SSRC identifier. Optionally, the BYE packet may include an 8-bit octet count followed by that many octets of text indicating the reason for leaving, e.g., "camera malfunction" or "RTP loop detected". The string has the same encoding as that described for SDES. If the string fills the packet to the next 32-bit boundary, the string is not null terminated. If not, the BYE packet is padded with null octets.

6.6 APP: Application-defined RTCP packet



The APP packet is intended for experimental use as new applications and new features are developed, without requiring packet type value registration. APP packets with unrecognized names should be ignored. After testing and if wider use is justified, it is recommended that each APP packet be redefined without the subtype and name fields and registered with the Internet Assigned Numbers Authority using an RTCP packet type.

version (V), padding (P), length:
As described for the SR packet (see Section 6.3.1).

subtype: 5 bits
May be used as a subtype to allow a set of APP packets to be defined under one unique name, or for any application-dependent data.

packet type (PT): 8 bits
Contains the constant 204 to identify this as an RTCP APP packet.

name: 4 octets
A name chosen by the person defining the set of APP packets to be unique with respect to other APP packets this application might receive. The application creator might choose to use the application name, and then coordinate the allocation of subtype values to others who want to define new packet types for the application. Alternatively, it is recommended that others choose a name based on the entity they represent, then coordinate the use of the name within that entity. The name is interpreted as a sequence of four ASCII characters, with uppercase and lowercase characters treated as distinct.

application-dependent data: variable length
Application-dependent data may or may not appear in an APP packet. It is interpreted by the application and not RTP itself. It must be a multiple of 32 bits long.

7. RTP Translators and Mixers

In addition to end systems, RTP supports the notion of "translators" and "mixers", which could be considered as "intermediate systems" at the RTP level. Although this support adds some complexity to the protocol, the need for these functions has been clearly established by experiments with multicast audio and video applications in the Internet. Example uses of translators and mixers given in Section 2.3 stem from the presence of firewalls and low bandwidth connections, both of which are likely to remain.

7.1 General Description

An RTP translator/mixer connects two or more transport-level "clouds". Typically, each cloud is defined by a common network and transport protocol (e.g., IP/UDP), multicast address or pair of unicast addresses, and transport level destination port. (Network-level protocol translators, such as IP version 4 to IP version 6, may be present within a cloud invisibly to RTP.) One system may serve as a translator or mixer for a number of RTP sessions, but each is considered a logically separate entity.

In order to avoid creating a loop when a translator or mixer is installed, the following rules must be observed:

- Each of the clouds connected by translators and mixers participating in one RTP session either must be distinct from all the others in at least one of these parameters (protocol, address, port), or must be isolated at the network level from the others.

- A derivative of the first rule is that there must not be multiple translators or mixers connected in parallel unless by some arrangement they partition the set of sources to be forwarded.

Similarly, all RTP end systems that can communicate through one or more RTP translators or mixers share the same SSRC space, that is, the SSRC identifiers must be unique among all these end systems. Section 8.2 describes the collision resolution algorithm by which SSRC identifiers are kept unique and loops are detected.

There may be many varieties of translators and mixers designed for different purposes and applications. Some examples are to add or remove encryption, change the encoding of the data or the underlying protocols, or replicate between a multicast address and one or more unicast addresses. The distinction between translators and mixers is that a translator passes through the data streams from different sources separately, whereas a mixer combines them to form one new stream:

Translator: Forwards RTP packets with their SSRC identifier intact; this makes it possible for receivers to identify individual sources even though packets from all the sources pass through the same translator and carry the translator's network source address. Some kinds of translators will pass through the data untouched, but others may change the encoding of the data and thus the RTP data payload type and timestamp. If multiple data packets are re-encoded into one, or vice versa, a translator must assign new sequence numbers to the outgoing packets. Losses in the incoming packet stream may induce corresponding gaps in the outgoing sequence numbers. Receivers cannot detect the presence of a translator unless they know by some other means what payload type or transport address was used by the original source.

Mixer: Receives streams of RTP data packets from one or more sources, possibly changes the data format, combines the streams in some manner and then forwards the combined stream. Since the timing among multiple input sources will not generally be synchronized, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream, so it is the synchronization source. Thus, all data packets forwarded by a mixer will be marked with the mixer's own SSRC identifier. In order to preserve the identity of the original sources contributing to the mixed packet, the mixer should insert their SSRC identifiers into the CSRC identifier list following the fixed RTP header of the packet. A mixer that is also itself a contributing source for some packet should explicitly include its own SSRC identifier in the CSRC list for that packet.

For some applications, it may be acceptable for a mixer not to identify sources in the CSRC list. However, this introduces the danger that loops involving those sources could not be detected.

The advantage of a mixer over a translator for applications like audio is that the output bandwidth is limited to that of one source even when multiple sources are active on the input side. This may be important for low-bandwidth links. The disadvantage is that receivers on the output side don't have any control over which sources are

passed through or muted, unless some mechanism is implemented for remote control of the mixer. The regeneration of synchronization information by mixers also means that receivers can't do inter-media synchronization of the original streams. A multi-media mixer could do it.

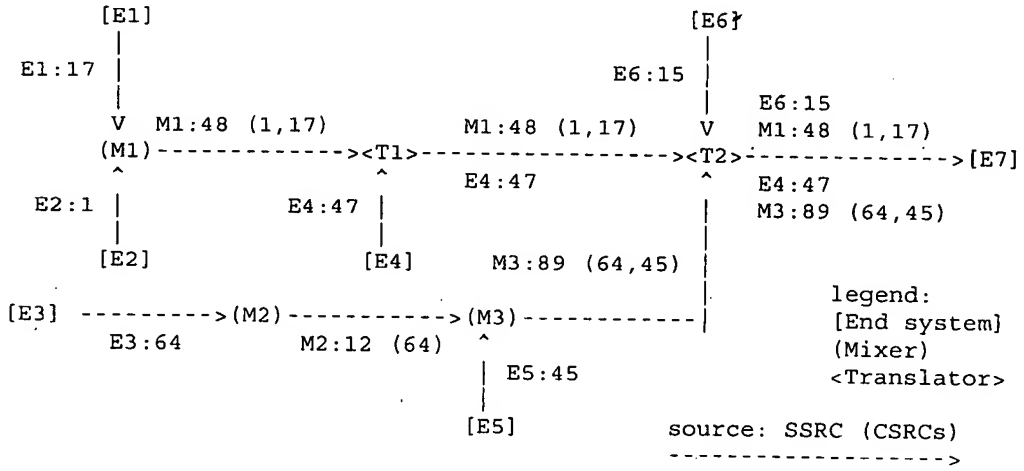


Figure 3: Sample RTP network with end systems, mixers and translators

A collection of mixers and translators is shown in Figure 3 to illustrate their effect on SSRC and CSRC identifiers. In the figure, end systems are shown as rectangles (named E), translators as triangles (named T) and mixers as ovals (named M). The notation "M1: 48(1,17)" designates a packet originating a mixer M1, identified with M1's (random) SSRC value of 48 and two CSRC identifiers, 1 and 17, copied from the SSRC identifiers of packets from E1 and E2.

7.2 RTCP Processing in Translators

In addition to forwarding data packets, perhaps modified, translators and mixers must also process RTCP packets. In many cases, they will take apart the compound RTCP packets received from end systems to aggregate SDES information and to modify the SR or RR packets. Retransmission of this information may be triggered by the packet arrival or by the RTCP interval timer of the translator or mixer itself.

A translator that does not modify the data packets, for example one that just replicates between a multicast address and a unicast address, may simply forward RTCP packets unmodified as well. A

translator that transforms the payload in some way must make corresponding transformations in the SR and RR information so that it still reflects the characteristics of the data and the reception quality. These translators must not simply forward RTCP packets. In general, a translator should not aggregate SR and RR packets from different sources into one packet since that would reduce the accuracy of the propagation delay measurements based on the LSR and DLSR fields.

SR sender information: A translator does not generate its own sender information, but forwards the SR packets received from one cloud to the others. The SSRC is left intact but the sender information must be modified if required by the translation. If a translator changes the data encoding, it must change the "sender's byte count" field. If it also combines several data packets into one output packet, it must change the "sender's packet count" field. If it changes the timestamp frequency, it must change the "RTP timestamp" field in the SR packet.

SR/RR reception report blocks: A translator forwards reception reports received from one cloud to the others. Note that these flow in the direction opposite to the data. The SSRC is left intact. If a translator combines several data packets into one output packet, and therefore changes the sequence numbers, it must make the inverse manipulation for the packet loss fields and the "extended last sequence number" field. This may be complex. In the extreme case, there may be no meaningful way to translate the reception reports, so the translator may pass on no reception report at all or a synthetic report based on its own reception. The general rule is to do what makes sense for a particular translation.

A translator does not require an SSRC identifier of its own, but may choose to allocate one for the purpose of sending reports about what it has received. These would be sent to all the connected clouds, each corresponding to the translation of the data stream as sent to that cloud, since reception reports are normally multicast to all participants.

SDES: Translators typically forward without change the SDES information they receive from one cloud to the others, but may, for example, decide to filter non-CNAME SDES information if bandwidth is limited. The CNAMEs must be forwarded to allow SSRC identifier collision detection to work. A translator that generates its own RR packets must send SDES CNAME information about itself to the same clouds that it sends those RR packets.

BYE: Translators forward BYE packets unchanged. Translators with their own SSRC should generate BYE packets with that SSRC identifier if they are about to cease forwarding packets.

APP: Translators forward APP packets unchanged.

7.3 RTCP Processing in Mixers

Since a mixer generates a new data stream of its own, it does not pass through SR or RR packets at all and instead generates new information for both sides.

SR sender information: A mixer does not pass through sender information from the sources it mixes because the characteristics of the source streams are lost in the mix. As a synchronization source, the mixer generates its own SR packets with sender information about the mixed data stream and sends them in the same direction as the mixed stream.

SR/RR reception report blocks: A mixer generates its own reception reports for sources in each cloud and sends them out only to the same cloud. It does not send these reception reports to the other clouds and does not forward reception reports from one cloud to the others because the sources would not be SSRCs there (only CSRCs).

SDES: Mixers typically forward without change the SDES information they receive from one cloud to the others, but may, for example, decide to filter non-CNAME SDES information if bandwidth is limited. The CNAMEs must be forwarded to allow SSRC identifier collision detection to work. (An identifier in a CSRC list generated by a mixer might collide with an SSRC identifier generated by an end system.) A mixer must send SDES CNAME information about itself to the same clouds that it sends SR or RR packets.

Since mixers do not forward SR or RR packets, they will typically be extracting SDES packets from a compound RTCP packet. To minimize overhead, chunks from the SDES packets may be aggregated into a single SDES packet which is then stacked on an SR or RR packet originating from the mixer. The RTCP packet rate may be different on each side of the mixer.

A mixer that does not insert CSRC identifiers may also refrain from forwarding SDES CNAMEs. In this case, the SSRC identifier spaces in the two clouds are independent. As mentioned earlier, this mode of operation creates a danger that loops can't be detected.

BYE: Mixers need to forward BYE packets. They should generate BYE packets with their own SSRC identifiers if they are about to cease forwarding packets.

APP: The treatment of APP packets by mixers is application-specific.

7.4 Cascaded Mixers

An RTP session may involve a collection of mixers and translators as shown in Figure 3. If two mixers are cascaded, such as M2 and M3 in the figure, packets received by a mixer may already have been mixed and may include a CSRC list with multiple identifiers. The second mixer should build the CSRC list for the outgoing packet using the CSRC identifiers from already-mixed input packets and the SSRC identifiers from unmixed input packets. This is shown in the output arc from mixer M3 labeled M3:89(64,45) in the figure. As in the case of mixers that are not cascaded, if the resulting CSRC list has more than 15 identifiers, the remainder cannot be included.

8. SSRC Identifier Allocation and Use

The SSRC identifier carried in the RTP header and in various fields of RTCP packets is a random 32-bit number that is required to be globally unique within an RTP session. It is crucial that the number be chosen with care in order that participants on the same network or starting at the same time are not likely to choose the same number.

It is not sufficient to use the local network address (such as an IPv4 address) for the identifier because the address may not be unique. Since RTP translators and mixers enable interoperation among multiple networks with different address spaces, the allocation patterns for addresses within two spaces might result in a much higher rate of collision than would occur with random allocation.

Multiple sources running on one host would also conflict.

It is also not sufficient to obtain an SSRC identifier simply by calling random() without carefully initializing the state. An example of how to generate a random identifier is presented in Appendix A.6.

8.1 Probability of Collision

Since the identifiers are chosen randomly, it is possible that two or more sources will choose the same number. Collision occurs with the highest probability when all sources are started simultaneously, for example when triggered automatically by some session management event. If N is the number of sources and L the length of the identifier (here, 32 bits), the probability that two sources

independently pick the same value can be approximated for large N [20] as 1 - exp(-N**2 / 2**(L+1)). For N=1000, the probability is roughly 10**-4.

The typical collision probability is much lower than the worst-case above. When one new source joins an RTP session in which all the other sources already have unique identifiers, the probability of collision is just the fraction of numbers used out of the space. Again, if N is the number of sources and L the length of the identifier, the probability of collision is N / 2**L. For N=1000, the probability is roughly 2*10**-7.

The probability of collision is further reduced by the opportunity for a new source to receive packets from other participants before sending its first packet (either data or control). If the new source keeps track of the other participants (by SSRC identifier), then before transmitting its first packet the new source can verify that its identifier does not conflict with any that have been received, or else choose again.

8.2 Collision Resolution and Loop Detection

Although the probability of SSRC identifier collision is low, all RTP implementations must be prepared to detect collisions and take the appropriate actions to resolve them. If a source discovers at any time that another source is using the same SSRC identifier as its own, it must send an RTCP BYE packet for the old identifier and choose another random one. If a receiver discovers that two other sources are colliding, it may keep the packets from one and discard the packets from the other when this can be detected by different source transport addresses or CNAMEs. The two sources are expected to resolve the collision so that the situation doesn't last.

Because the random identifiers are kept globally unique for each RTP session, they can also be used to detect loops that may be introduced by mixers or translators. A loop causes duplication of data and control information, either unmodified or possibly mixed, as in the following examples:

- o A translator may incorrectly forward a packet to the same multicast group from which it has received the packet, either directly or through a chain of translators. In that case, the same packet appears several times, originating from different network sources.

- o Two translators incorrectly set up in parallel, i.e., with the same multicast groups on both sides, would both forward packets from one multicast group to the other. Unidirectional

translators would produce two copies; bidirectional translators would form a loop.

- o A mixer can close a loop by sending to the same transport destination upon which it receives packets, either directly or through another mixer or translator. In this case a source might show up both as an SSRC on a data packet and a CSRC in a mixed data packet.

A source may discover that its own packets are being looped, or that packets from another source are being looped (a third-party loop).

Both loops and collisions in the random selection of a source identifier result in packets arriving with the same SSRC identifier but a different source transport address, which may be that of the end system originating the packet or an intermediate system. Consequently, if a source changes its source transport address, it must also choose a new SSRC identifier to avoid being interpreted as a looped source. Loops or collisions occurring on the far side of a translator or mixer cannot be detected using the source transport address if all copies of the packets go through the translator or mixer, however collisions may still be detected when chunks from two RTCP SDES packets contain the same SSRC identifier but different CNAMEs.

To detect and resolve these conflicts, an RTP implementation must include an algorithm similar to the one described below. It ignores packets from a new source or loop that collide with an established source. It resolves collisions with the participant's own SSRC identifier by sending an RTCP BYE for the old identifier and choosing a new one. However, when the collision was induced by a loop of the participant's own packets, the algorithm will choose a new identifier only once and thereafter ignore packets from the looping source transport address. This is required to avoid a flood of BYE packets.

This algorithm depends upon the source transport address being the same for both RTP and RTCP packets from a source. The algorithm would require modifications to support applications that don't meet this constraint.

This algorithm requires keeping a table indexed by source identifiers and containing the source transport address from which the identifier was (first) received, along with other state for that source. Each SSRC or CSRC identifier received in a data or control packet is looked up in this table in order to process that data or control information. For control packets, each element with its own SSRC, for example an SDES chunk, requires a separate lookup. (The SSRC in a reception report block is an exception.) If the SSRC or CSRC is not

found, a new entry is created. These table entries are removed when an RTCP BYE packet is received with the corresponding SSRC, or after no packets have arrived for a relatively long time (see Section 6.2.1).

In order to track loops of the participant's own data packets, it is also necessary to keep a separate list of source transport addresses (not identifiers) that have been found to be conflicting. Note that this should be a short list, usually empty. Each element in this list stores the source address plus the time when the most recent conflicting packet was received. An element may be removed from the list when no conflicting packet has arrived from that source for a time on the order of 10 RTCP report intervals (see Section 6.2).

For the algorithm as shown, it is assumed that the participant's own source identifier and state are included in the source identifier table. The algorithm could be restructured to first make a separate comparison against the participant's own source identifier.

```
    IF the SSRC or CSRC identifier is not found in the source
       identifier table:
    THEN create a new entry storing the source transport address
         and the SSRC or CSRC along with other state.
         CONTINUE with normal processing.

    (identifier is found in the table)

    IF the source transport address from the packet matches
       the one saved in the table entry for this identifier:
    THEN CONTINUE with normal processing.

    (an identifier collision or a loop is indicated)

    IF the source identifier is not the participant's own:
    THEN IF the source identifier is from an RTCP SDES chunk
            containing a CNAME item that differs from the CNAME
            in the table entry:
         THEN (optionally) count a third-party collision.
         ELSE (optionally) count a third-party loop.
         ABORT processing of data packet or control element.

    (a collision or loop of the participant's own data)

    IF the source transport address is found in the list of
      conflicting addresses:
    THEN IF the source identifier is not from an RTCP SDES chunk
            containing a CNAME item OR if that CNAME is the
            participant's own:
```

```
            THEN (optionally) count occurrence of own traffic looped.
                 mark current time in conflicting address list entry.
                 ABORT processing of data packet or control element.
        log occurrence of a collision.
        create a new entry in the conflicting address list and
        mark current time.
        send an RTCP BYE packet with the old SSRC identifier.
        choose a new identifier.
        create a new entry in the source identifier table with the
          old SSRC plus the source transport address from the packet
          being processed.
        CONTINUE with normal processing.
```

In this algorithm, packets from a newly conflicting source address will be ignored and packets from the original source will be kept. (If the original source was through a mixer and later the same source is received directly, the receiver may be well advised to switch unless other sources in the mix would be lost.) If no packets arrive from the original source for an extended period, the table entry will be timed out and the new source will be able to take over. This might occur if the original source detects the collision and moves to a new source identifier, but in the usual case an RTCP BYE packet will be received from the original source to delete the state without having to wait for a timeout.

When a new SSRC identifier is chosen due to a collision, the candidate identifier should first be looked up in the source identifier table to see if it was already in use by some other source. If so, another candidate should be generated and the process repeated.

A loop of data packets to a multicast destination can cause severe network flooding. All mixers and translators are required to implement a loop detection algorithm like the one here so that they can break loops. This should limit the excess traffic to no more than one duplicate copy of the original traffic, which may allow the session to continue so that the cause of the loop can be found and fixed. However, in extreme cases where a mixer or translator does not properly break the loop and high traffic levels result, it may be necessary for end systems to cease transmitting data or control packets entirely. This decision may depend upon the application. An error condition should be indicated as appropriate. Transmission might be attempted again periodically after a long, random time (on the order of minutes).

9. Security

Lower layer protocols may eventually provide all the security services that may be desired for applications of RTP, including authentication, integrity, and confidentiality. These services have recently been specified for IP. Since the need for a confidentiality service is well established in the initial audio and video applications that are expected to use RTP, a confidentiality service is defined in the next section for use with RTP and RTCP until lower layer services are available. The overhead on the protocol for this service is low, so the penalty will be minimal if this service is obsoleted by lower layer services in the future.

Alternatively, other services, other implementations of services and other algorithms may be defined for RTP in the future if warranted. The selection presented here is meant to simplify implementation of interoperable, secure applications and provide guidance to implementors. No claim is made that the methods presented here are appropriate for a particular security need. A profile may specify which services and algorithms should be offered by applications, and may provide guidance as to their appropriate use.

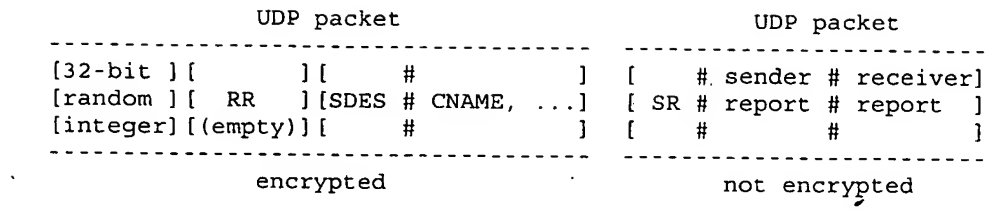
Key distribution and certificates are outside the scope of this document.

9.1 Confidentiality

Confidentiality means that only the intended receiver(s) can decode the received packets; for others, the packet contains no useful information. Confidentiality of the content is achieved by encryption.

When encryption of RTP or RTCP is desired, all the octets that will be encapsulated for transmission in a single lower-layer packet are encrypted as a unit. For RTCP, a 32-bit random number is prepended to the unit before encryption to deter known plaintext attacks. For RTP, no prefix is required because the sequence number and timestamp fields are initialized with random offsets.

For RTCP, it is allowed to split a compound RTCP packet into two lower-layer packets, one to be encrypted and one to be sent in the clear. For example, SDES information might be encrypted while reception reports were sent in the clear to accommodate third-party monitors that are not privy to the encryption key. In this example, depicted in Fig. 4, the SDES information must be appended to an RR packet with no reports (and the encrypted) to satisfy the requirement that all compound RTCP packets begin with an SR or RR packet.



#: SSRC

Figure 4: Encrypted and non-encrypted RTCP packets

The presence of encryption and the use of the correct key are confirmed by the receiver through header or payload validity checks. Examples of such validity checks for RTP and RTCP headers are given in Appendices A.1 and A.2.

The default encryption algorithm is the Data Encryption Standard (DES) algorithm in cipher block chaining (CBC) mode, as described in Section 1.1 of RFC 1423 [21], except that padding to a multiple of 8 octets is indicated as described for the P bit in Section 5.1. The initialization vector is zero because random values are supplied in the RTP header or by the random prefix for compound RTCP packets. For details on the use of CBC initialization vectors, see [22]. Implementations that support encryption should always support the DES algorithm in CBC mode as the default to maximize interoperability. This method is chosen because it has been demonstrated to be easy and practical to use in experimental audio and video tools in operation on the Internet. Other encryption algorithms may be specified dynamically for a session by non-RTP means.

As an alternative to encryption at the RTP level as described above, profiles may define additional payload types for encrypted encodings. Those encodings must specify how padding and other aspects of the encryption should be handled. This method allows encrypting only the data while leaving the headers in the clear for applications where that is desired. It may be particularly useful for hardware devices that will handle both decryption and decoding.

9.2 Authentication and Message Integrity

Authentication and message integrity are not defined in the current specification of RTP since these services would not be directly feasible without a key management infrastructure. It is expected that authentication and integrity services will be provided by lower layer protocols in the future.

10. RTP over Network and Transport Protocols

This section describes issues specific to carrying RTP packets within particular network and transport protocols. The following rules apply unless superseded by protocol-specific definitions outside this specification.

RTP relies on the underlying protocol(s) to provide demultiplexing of RTP data and RTCP control streams. For UDP and similar protocols, RTP uses an even port number and the corresponding RTCP stream uses the next higher (odd) port number. If an application is supplied with an odd number for use as the RTP port, it should replace this number with the next lower (even) number.

RTP data packets contain no length field or other delineation, therefore RTP relies on the underlying protocol(s) to provide a length indication. The maximum length of RTP packets is limited only by the underlying protocols.

If RTP packets are to be carried in an underlying protocol that provides the abstraction of a continuous octet stream rather than messages (packets), an encapsulation of the RTP packets must be defined to provide a framing mechanism. Framing is also needed if the underlying protocol may contain padding so that the extent of the RTP payload cannot be determined. The framing mechanism is not defined here.

A profile may specify a framing method to be used even when RTP is carried in protocols that do provide framing in order to allow carrying several RTP packets in one lower-layer protocol data unit, such as a UDP packet. Carrying several RTP packets in one network or transport packet reduces header overhead and may simplify synchronization between different streams.

11. Summary of Protocol Constants

This section contains a summary listing of the constants defined in this specification.

The RTP payload type (PT) constants are defined in profiles rather than this document. However, the octet of the RTP header which contains the marker bit(s) and payload type must avoid the reserved values 200 and 201 (decimal) to distinguish RTP packets from the RTCP SR and RR packet types for the header validation procedure described in Appendix A.1. For the standard definition of one marker bit and a 7-bit payload type field as shown in this specification, this restriction means that payload types 72 and 73 are reserved.

11.1 RTCP packet types

abbrev.	name	value
SR	sender report	200
RR	receiver report	201
SDES	source description	202
BYE	goodbye	203
APP	application-defined	204

These type values were chosen in the range 200-204 for improved header validity checking of RTCP packets compared to RTP packets or other unrelated packets. When the RTCP packet type field is compared to the corresponding octet of the RTP header, this range corresponds to the marker bit being 1 (which it usually is not in data packets) and to the high bit of the standard payload type field being 1 (since the static payload types are typically defined in the low half). This range was also chosen to be some distance numerically from 0 and 255 since all-zeros and all-ones are common data patterns.

Since all compound RTCP packets must begin with SR or RR, these codes were chosen as an even/odd pair to allow the RTCP validity check to test the maximum number of bits with mask and value.

Other constants are assigned by IANA. Experimenters are encouraged to register the numbers they need for experiments, and then unregister those which prove to be unneeded.

11.2 SDES types

abbrev.	name	value
END	end of SDES list	0
CNAME	canonical name	1
NAME	user name	2
EMAIL	user's electronic mail address	3
PHONE	user's phone number	4
LOC	geographic user location	5
TOOL	name of application or tool	6
NOTE	notice about the source	7
PRIV	private extensions	8

Other constants are assigned by IANA. Experimenters are encouraged to register the numbers they need for experiments, and then unregister those which prove to be unneeded.

12. RTP Profiles and Payload Format Specifications

A complete specification of RTP for a particular application will require one or more companion documents of two types described here: profiles, and payload format specifications.

RTP may be used for a variety of applications with somewhat differing requirements. The flexibility to adapt to those requirements is provided by allowing multiple choices in the main protocol specification, then selecting the appropriate choices or defining extensions for a particular environment and class of applications in a separate profile document. Typically an application will operate under only one profile so there is no explicit indication of which profile is in use. A profile for audio and video applications may be found in the companion Internet-Draft draft-ietf-avt-profile for

The second type of companion document is a payload format specification, which defines how a particular kind of payload data, such as H.261 encoded video, should be carried in RTP. These documents are typically titled "RTP Payload Format for XYZ Audio/Video Encoding". Payload formats may be useful under multiple profiles and may therefore be defined independently of any particular profile. The profile documents are then responsible for assigning a default mapping of that format to a payload type value if needed.

Within this specification, the following items have been identified for possible definition within a profile, but this list is not meant to be exhaustive:

RTP data header: The octet in the RTP data header that contains the marker bit and payload type field may be redefined by a profile to suit different requirements, for example with more or fewer marker bits (Section 5.3).

Payload types: Assuming that a payload type field is included, the profile will usually define a set of payload formats (e.g., media encodings) and a default static mapping of those formats to payload type values. Some of the payload formats may be defined by reference to separate payload format specifications. For each payload type defined, the profile must specify the RTP timestamp clock rate to be used (Section 5.1).

RTP data header additions: Additional fields may be appended to the fixed RTP data header if some additional functionality is required across the profile's class of applications independent of payload type (Section 5.3).

RTP data header extensions: The contents of the first 16 bits of the RTP data header extension structure must be defined if use of that mechanism is to be allowed under the profile for implementation-specific extensions (Section 5.3.1).

RTCP packet types: New application-class-specific RTCP packet types may be defined and registered with IANA.

RTCP report interval: A profile should specify that the values suggested in Section 6.2 for the constants employed in the calculation of the RTCP report interval will be used. Those are the RTCP fraction of session bandwidth, the minimum report interval, and the bandwidth split between senders and receivers. A profile may specify alternate values if they have been demonstrated to work in a scalable manner.

SR/RR extension: An extension section may be defined for the RTCP SR and RR packets if there is additional information that should be reported regularly about the sender or receivers (Section 6.3.3).

SDES use: The profile may specify the relative priorities for RTCP SDES items to be transmitted or excluded entirely (Section 6.2.2); an alternate syntax or semantics for the CNAME item (Section 6.4.1); the format of the LOC item (Section 6.4.5); the semantics and use of the NOTE item (Section 6.4.7); or new SDES item types to be registered with IANA.

Security: A profile may specify which security services and algorithms should be offered by applications, and may provide guidance as to their appropriate use (Section 9).

String-to-key mapping: A profile may specify how a user-provided password or pass phrase is mapped into an encryption key.

Underlying protocol: Use of a particular underlying network or transport layer protocol to carry RTP packets may be required.

Transport mapping: A mapping of RTP and RTCP to transport-level addresses, e.g., UDP ports, other than the standard mapping defined in Section 10 may be specified.

Encapsulation: An encapsulation of RTP packets may be defined to allow multiple RTP data packets to be carried in one lower-layer packet or to provide framing over underlying protocols that do not already do so (Section 10).

It is not expected that a new profile will be required for every application. Within one application class, it would be better to extend an existing profile rather than make a new one in order to facilitate interoperation among the applications since each will typically run under only one profile. Simple extensions such as the definition of additional payload type values or RTCP packet types may be accomplished by registering them through the Internet Assigned Numbers Authority and publishing their descriptions in an addendum to the profile or in a payload format specification.

A. Algorithms

We provide examples of C code for aspects of RTP sender and receiver algorithms. There may be other implementation methods that are faster in particular operating environments or have other advantages. These implementation notes are for informational purposes only and are meant to clarify the RTP specification.

The following definitions are used for all examples; for clarity and brevity, the structure definitions are only valid for 32-bit big-endian (most significant octet first) architectures. Bit fields are assumed to be packed tightly in big-endian bit order, with no additional padding. Modifications would be required to construct a portable implementation.

```
/*
 * rtp.h  --  RTP header file (RFC XXXX)
 */
#include <sys/types.h>

/*
 * The type definitions below are valid for 32-bit architectures and
 * may have to be adjusted for 16- or 64-bit architectures.
 */
typedef unsigned char  u_int8;
typedef unsigned short u_int16;
typedef unsigned int   u_int32;
typedef          short int16;

/*
 * Current protocol version.
 */
#define RTP_VERSION    2

#define RTP_SEQ_MOD (1<<16)
#define RTP_MAX_SDES 255      /* maximum text length for SDES */

typedef enum {
    RTCP_SR   = 200,
    RTCP_RR   = 201,
    RTCP_SDES = 202,
    RTCP_BYE  = 203,
    RTCP_APP  = 204
} rtcp_type_t;

typedef enum {
    RTCP_SDES_END   = 0,
    RTCP_SDES_CNAME = 1,
```

```
        RTCP_SDES_NAME  = 2,
        RTCP_SDES_EMAIL = 3,
        RTCP_SDES_PHONE = 4,
        RTCP_SDES_LOC   = 5,
        RTCP_SDES_TOOL  = 6,
        RTCP_SDES_NOTE  = 7,
        RTCP_SDES_PRIV  = 8
    } rtcp_sdes_type_t;

    /*
     * RTP data header
     */
    typedef struct {
        unsigned int version:2;   /* protocol version */
        unsigned int p:1;         /* padding flag */
        unsigned int x:1;         /* header extension flag */
        unsigned int cc:4;        /* CSRC count */
        unsigned int m:1;         /* marker bit */
        unsigned int pt:7;        /* payload type */
        u_int16 seq;              /* sequence number */
        u_int32 ts;               /* timestamp */
        u_int32 ssrc;             /* synchronization source */
        u_int32 csrc[1];          /* optional CSRC list */
    } rtp_hdr_t;

    /*
     * RTCP common header word
     */
    typedef struct {
        unsigned int version:2;   /* protocol version */
        unsigned int p:1;         /* padding flag */
        unsigned int count:5;     /* varies by packet type */
        unsigned int pt:8;        /* RTCP packet type */
        u_int16 length;           /* pkt len in words, w/o this word */
    } rtcp_common_t;

    /*
     * Big-endian mask for version, padding bit and packet type pair
     */
    #define RTCP_VALID_MASK (0xc000 | 0x2000 | 0xfe)
    #define RTCP_VALID_VALUE ((RTP_VERSION << 14) | RTCP_SR)

    /*
     * Reception report block
     */
    typedef struct {
        u_int32 ssrc;             /* data source being reported */
        unsigned int fraction:8;  /* fraction lost since last SR/RR */
```

```
        int lost:24;            /* cumul. no. pkts lost (signed!) */
        u_int32 last_seq;       /* extended last seq. no. received */
        u_int32 jitter;         /* interarrival jitter */
        u_int32 lsr;            /* last SR packet from this source */
        u_int32 dlsr;           /* delay since last SR packet */
    } rtcp_rr_t;

    /*
     * SDES item
     */
    typedef struct {
        u_int8 type;            /* type of item (rtcp_sdes_type_t) */
        u_int8 length;          /* length of item (in octets) */
        char data[1];           /* text, not null-terminated */
    } rtcp_sdes_item_t;

    /*
     * One RTCP packet
     */
    typedef struct {
        rtcp_common_t common;   /* common header */
        union {
            /* sender report (SR) */
            struct {
                u_int32 ssrc;     /* sender generating this report */
                u_int32 ntp_sec;  /* NTP timestamp */
                u_int32 ntp_frac;
                u_int32 rtp_ts;   /* RTP timestamp */
                u_int32 psent;    /* packets sent */
                u_int32 osent;    /* octets sent */
                rtcp_rr_t rr[1];  /* variable-length list */
            } sr;

            /* reception report (RR) */
            struct {
                u_int32 ssrc;     /* receiver generating this report */
                rtcp_rr_t rr[1];  /* variable-length list */
            } rr;

            /* source description (SDES) */
            struct rtcp_sdes {
                u_int32 src;      /* first SSRC/CSRC */
                rtcp_sdes_item_t item[1]; /* list of SDES items */
            } sdes;

            /* BYE */
            struct {
                u_int32 src[1];   /* list of sources */
```

```
            /* can't express trailing text for reason */
        } bye;
    } r;
} rtcp_t;

typedef struct rtcp_sdes rtcp_sdes_t;

/*
 * Per-source state information
 */
typedef struct {
    u_int16 max_seq;        /* highest seq. number seen */
    u_int32 cycles;         /* shifted count of seq. number cycles */
    u_int32 base_seq;       /* base seq number */
    u_int32 bad_seq;        /* last 'bad' seq number + 1 */
    u_int32 probation;      /* sequ. packets till source is valid */
    u_int32 received;       /* packets received */
    u_int32 expected_prior; /* packet expected at last interval */
    u_int32 received_prior; /* packet received at last interval */
    u_int32 transit;        /* relative trans time for prev pkt */
    u_int32 jitter;         /* estimated jitter */
    /* ... */
} source;
```

A.1 RTP Data Header Validity Checks

An RTP receiver should check the validity of the RTP header on incoming packets since they might be encrypted or might be from a different application that happens to be misaddressed. Similarly, if encryption is enabled, the header validity check is needed to verify that incoming packets have been correctly decrypted, although a failure of the header validity check (e.g., unknown payload type) may not necessarily indicate decryption failure.

Only weak validity checks are possible on an RTP data packet from a source that has not been heard before:

- o RTP version field must equal 2.
- o The payload type must be known, in particular it must not be equal to SR or RR.
- o If the P bit is set, then the last octet of the packet must contain a valid octet count, in particular, less than the total packet length minus the header size.
- o The X bit must be zero if the profile does not specify that the header extension mechanism may be used. Otherwise, the

extension length field must be less than the total packet size minus the fixed header length and padding.

- The length of the packet must be consistent with CC and payload type (if payloads have a known length).

The last three checks are somewhat complex and not always possible, leaving only the first two which total just a few bits. If the SSRC identifier in the packet is one that has been received before, then the packet is probably valid and checking if the sequence number is in the expected range provides further validation. If the SSRC identifier has not been seen before, then data packets carrying that identifier may be considered invalid until a small number of them arrive with consecutive sequence numbers.

The routine update_seq shown below ensures that a source is declared valid only after MIN_SEQUENTIAL packets have been received in sequence. It also validates the sequence number seq of a newly received packet and updates the sequence state for the packet's source in the structure to which s points.

When a new source is heard for the first time, that is, its SSRC identifier is not in the table (see Section 8.2), and the per-source state is allocated for it, s->probation should be set to the number of sequential packets required before declaring a source valid (parameter MIN_SEQUENTIAL) and s->max_seq initialized to seq-1 s->probation marks the source as not yet valid so the state may be discarded after a short timeout rather than a long one, as discussed in Section 6.2.1.

After a source is considered valid, the sequence number is considered valid if it is no more than MAX_DROPOUT ahead of s->max_seq nor more than MAX_MISORDER behind. If the new sequence number is ahead of max_seq modulo the RTP sequence number range (16 bits), but is smaller than max_seq , it has wrapped around and the (shifted) count of sequence number cycles is incremented. A value of one is returned to indicate a valid sequence number.

Otherwise, the value zero is returned to indicate that the validation failed, and the bad sequence number is stored. If the next packet received carries the next higher sequence number, it is considered the valid start of a new packet sequence presumably caused by an extended dropout or a source restart. Since multiple complete sequence number cycles may have been missed, the packet loss statistics are reset.

Typical values for the parameters are shown, based on a maximum misordering time of 2 seconds at 50 packets/second and a maximum

dropout of 1 minute. The dropout parameter MAX_DROPOUT should be a small fraction of the 16-bit sequence number space to give a reasonable probability that new sequence numbers after a restart will not fall in the acceptable range for sequence numbers from before the restart.

```
void init_seq(source *s, u_int16 seq)
{
    s->base_seq = seq - 1;
    s->max_seq = seq;
    s->bad_seq = RTP_SEQ_MOD + 1;
    s->cycles = 0;
    s->received = 0;
    s->received_prior = 0;
    s->expected_prior = 0;
    /* other initialization */
}

int update_seq(source *s, u_int16 seq)
{
    u_int16 udelta = seq - s->max_seq;
    const int MAX_DROPOUT = 3000;
    const int MAX_MISORDER = 100;
    const int MIN_SEQUENTIAL = 2;

    /*
     * Source is not valid until MIN_SEQUENTIAL packets with
     * sequential sequence numbers have been received.
     */
    if (s->probation) {
        /* packet is in sequence */
        if (seq == s->max_seq + 1) {
            s->probation--;
            s->max_seq = seq;
            if (s->probation == 0) {
                init_seq(s, seq);
                s->received++;
                return 1;
            }
        } else {
            s->probation = MIN_SEQUENTIAL - 1;
            s->max_seq = seq;
        }
        return 0;
    } else if (udelta < MAX_DROPOUT) {
        /* in order, with permissible gap */
        if (seq < s->max_seq) {
            /*
```

```
             * Sequence number wrapped - count another 64K cycle.
             */
            s->cycles += RTP_SEQ_MOD;
        }
        s->max_seq = seq;
    } else if (udelta <= RTP_SEQ_MOD - MAX_MISORDER) {
        /* the sequence number made a very large jump */
        if (seq == s->bad_seq) {
            /*
             * Two sequential packets -- assume that the other side
             * restarted without telling us so just re-sync
             * (i.e., pretend this was the first packet).
             */
            init_seq(s, seq);
        }
        else {
            s->bad_seq = (seq + 1) & (RTP_SEQ_MOD-1);
            return 0;
        }
    } else {
        /* duplicate or reordered packet */
    }
    s->received++;
    return 1;
}
```

The validity check can be made stronger requiring more than two packets in sequence. The disadvantages are that a larger number of initial packets will be discarded and that high packet loss rates could prevent validation. However, because the RTCP header validation is relatively strong, if an RTCP packet is received from a source before the data packets, the count could be adjusted so that only two packets are required in sequence. If initial data loss for a few seconds can be tolerated, an application could choose to discard all data packets from a source until a valid RTCP packet has been received from that source.

Depending on the application and encoding, algorithms may exploit additional knowledge about the payload format for further validation. For payload types where the timestamp increment is the same for all packets, the timestamp values can be predicted from the previous packet received from the same source using the sequence number difference (assuming no change in payload type).

A strong "fast-path" check is possible since with high probability the first four octets in the header of a newly received RTP data packet will be just the same as that of the previous packet from the same SSRC except that the sequence number will have increased by one.

Similarly, a single-entry cache may be used for faster SSRC lookups in applications where data is typically received from one source at a time.

A.2 RTCP Header Validity Checks

The following checks can be applied to RTCP packets.

- o RTP version field must equal 2.
- o The payload type field of the first RTCP packet in a compound packet must be equal to SR or RR.
- o The padding bit (P) should be zero for the first packet of a compound RTCP packet because only the last should possibly need padding.
- o The length fields of the individual RTCP packets must total to the overall length of the compound RTCP packet as received. This is a fairly strong check.

The code fragment below performs all of these checks. The packet type is not checked for subsequent packets since unknown packet types may be present and should be ignored.

```
u_int32 len;        /* length of compound RTCP packet in words */
rtcp_t *r;          /* RTCP header */
rtcp_t *end;        /* end of compound RTCP packet */

if ((*(u_int16 *)r & RTCP_VALID_MASK) != RTCP_VALID_VALUE) {
    /* something wrong with packet format */
}
end = (rtcp_t *)((u_int32 *)r + len);

do r = (rtcp_t *)((u_int32 *)r + r->common.length + 1);
while (r < end && r->common.version == 2);

if (r != end) {
    /* something wrong with packet format */
}
```

A.3 Determining the Number of RTP Packets Expected and Lost

In order to compute packet loss rates, the number of packets expected and actually received from each source needs to be known, using per-source state information defined in struct source referenced via pointer s in the code below. The number of packets received is simply the count of packets as they arrive, including any late or duplicate

packets. The number of packets expected can be computed by the receiver as the difference between the highest sequence number received (s->max_seq) and the first sequence number received (s->base_seq). Since the sequence number is only 16 bits and will wrap around, it is necessary to extend the highest sequence number with the (shifted) count of sequence number wraparounds (s->cycles). Both the received packet count and the count of cycles are maintained the RTP header validity check routine in Appendix A.1.

```
extended_max = s->cycles + s->max_seq;
expected = extended_max - s->base_seq + 1;
```

The number of packets lost is defined to be the number of packets expected less the number of packets actually received:

```
lost = expected - s->received;
```

Since this number is carried in 24 bits, it should be clamped at 0xffffff rather than wrap around to zero.

The fraction of packets lost during the last reporting interval (since the previous SR or RR packet was sent) is calculated from differences in the expected and received packet counts across the interval, where expected_prior and received_prior are the values saved when the previous reception report was generated:

```
expected_interval = expected - s->expected_prior;
s->expected_prior = expected;
received_interval = s->received - s->received_prior;
s->received_prior = s->received;
lost_interval = expected_interval - received_interval;
if (expected_interval == 0 || lost_interval <= 0) fraction = 0;
else fraction = (lost_interval << 8) / expected_interval;
```

The resulting fraction is an 8-bit fixed point number with the binary point at the left edge.

A.4 Generating SDES RTCP Packets

This function builds one SDES chunk into buffer b composed of argc items supplied in arrays type , value and length b

```
char *rtp_write_sdes(char *b, u_int32 src, int argc,
                     rtcp_sdes_type_t type[], char *value[],
                     int length[])
{
    rtcp_sdes_t *s = (rtcp_sdes_t *)b;
    rtcp_sdes_item_t *rsp;
```

```
        int i;
        int len;
        int pad;

        /* SSRC header */
        s->src = src;
        rsp = &s->item[0];

        /* SDES items */
        for (i = 0; i < argc; i++) {
            rsp->type = type[i];
            len = length[i];
            if (len > RTP_MAX_SDES) {
                /* invalid length, may want to take other action */
                len = RTP_MAX_SDES;
            }
            rsp->length = len;
            memcpy(rsp->data, value[i], len);
            rsp = (rtcp_sdes_item_t *)&rsp->data[len];
        }

        /* terminate with end marker and pad to next 4-octet boundary */
        len = ((char *) rsp) - b;
        pad = 4 - (len & 0x3);
        b = (char *) rsp;
        while (pad--) *b++ = RTCP_SDES_END;

        return b;
    }
```

A.5 Parsing RTCP SDES Packets

This function parses an SDES packet, calling functions find_member() to find a pointer to the information for a session member given the SSRC identifier and member_sdes() to store the new SDES information for that member. This function expects a pointer to the header of the RTCP packet.

```
    void rtp_read_sdes(rtcp_t *r)
    {
        int count = r->common.count;
        rtcp_sdes_t *sd = &r->r.sdes;
        rtcp_sdes_item_t *rsp, *rspn;
        rtcp_sdes_item_t *end = (rtcp_sdes_item_t *)
                                ((u_int32 *)r + r->common.length + 1);
        source *s;

        while (--count >= 0) {
```

```
            rsp = &sd->item[0];
            if (rsp >= end) break;
            s = find_member(sd->src);

            for (; rsp->type; rsp = rspn ) {
                rspn = (rtcp_sdes_item_t *)((char*)rsp+rsp->length+2);
                if (rspn >= end) {
                    rsp = rspn;
                    break;
                }
                member_sdes(s, rsp->type, rsp->data, rsp->length);
            }
            sd = (rtcp_sdes_t *)
                 ((u_int32 *)sd + (((char *)rsp - (char *)sd) >> 2)+1);
        }
        if (count >= 0) {
            /* invalid packet format */
        }
    }
```

A.6 Generating a Random 32-bit Identifier

The following subroutine generates a random 32-bit identifier using the MD5 routines published in RFC 1321 [23]. The system routines may not be present on all operating systems, but they should serve as hints as to what kinds of information may be used. Other system calls that may be appropriate include

- o getdomainname() ,
- o getwd() , or
- o getrusage()

"Live" video or audio samples are also a good source of random numbers, but care must be taken to avoid using a turned-off microphone or blinded camera as a source [7].

Use of this or similar routine is suggested to generate the initial seed for the random number generator producing the RTCP period (as shown in Appendix A.7), to generate the initial values for the sequence number and timestamp, and to generate SSRC values. Since this routine is likely to be CPU-intensive, its direct use to generate RTCP periods is inappropriate because predictability is not an issue. Note that this routine produces the same result on repeated calls until the value of the system clock changes unless different values are supplied for the type argument.

```
/*
 * Generate a random 32-bit quantity.
 */
#include <sys/types.h>    /* u_long */
#include <sys/time.h>     /* gettimeofday() */
#include <unistd.h>       /* get..() */
#include <stdio.h>        /* printf() */
#include <time.h>         /* clock() */
#include <sys/utsname.h>  /* uname() */
#include "global.h"       /* from RFC 1321 */
#include "md5.h"          /* from RFC 1321 */

#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final

static u_long md_32(char *string, int length)
{
    MD_CTX context;
    union {
        char   c[16];
        u_long x[4];
    } digest;
    u_long r;
    int i;

    MDInit (&context);
    MDUpdate (&context, string, length);
    MDFinal ((unsigned char *)&digest, &context);
    r = 0;
    for (i = 0; i < 3; i++) {
        r ^= digest.x[i];
    }
    return r;
}                                 /* md_32 */


/*
 * Return random unsigned 32-bit quantity. Use 'type' argument if you
 * need to generate several different values in close succession.
 */
u_int32 random32(int type)
{
    struct {
        int     type;
        struct  timeval tv;
        clock_t cpu;
```

```
        pid_t   pid;
        u_long  hid;
        uid_t   uid;
        gid_t   gid;
        struct  utsname name;
    } s;

    gettimeofday(&s.tv, 0);
    uname(&s.name);
    s.type = type;
    s.cpu  = clock();
    s.pid  = getpid();
    s.hid  = gethostid();
    s.uid  = getuid();
    s.gid  = getgid();

    return md_32((char *)&s, sizeof(s));
}                                   /* random32 */
```

A.7 Computing the RTCP Transmission Interval

The following function returns the time between transmissions of RTCP packets, measured in seconds. It should be called after sending one compound RTCP packet to calculate the delay until the next should be sent. This function should also be called to calculate the delay before sending the first RTCP packet upon startup rather than send the packet immediately. This avoids any burst of RTCP packets if an application is started at many sites simultaneously, for example as a result of a session announcement.

The parameters have the following meaning:

rtcp_bw: The target RTCP bandwidth, i.e., the total bandwidth that will be used for RTCP packets by all members of this session, in octets per second. This should be 5% of the "session bandwidth" parameter supplied to the application at startup.

senders: Number of active senders since sending last report, known from construction of receiver reports for this RTCP packet. Includes ourselves, if we also sent during this interval.

members: The estimated number of session members, including ourselves. Incremented as we discover new session members from the receipt of RTP or RTCP packets, and decremented as session members leave (via RTCP BYE) or their state is timed out (30 minutes is recommended). On the first call, this parameter should have the value 1.

we_sent: Flag that is true if we have sent data during the last two RTCP intervals. If the flag is true, the compound RTCP packet just sent contained an SR packet.

packet_size: The size of the compound RTCP packet just sent, in octets, including the network encapsulation (e.g., 28 octets for UDP over IP).

avg_rtcp_size: Pointer to estimator for compound RTCP packet size; initialized and updated by this function for the packet just sent, and also updated by an identical line of code in the RTCP receive routine for every RTCP packet received from other participants in the session.

initial: Flag that is true for the first call upon startup to calculate the time until the first report should be sent.

```
#include <math.h>

double rtcp_interval(int members,
                     int senders,
                     double rtcp_bw,
                     int we_sent,
                     int packet_size,
                     int *avg_rtcp_size,
                     int initial)
{
    /*
     * Minimum time between RTCP packets from this site (in seconds).
     * This time prevents the reports from 'clumping' when sessions
     * are small and the law of large numbers isn't helping to smooth
     * out the traffic.  It also keeps the report interval from
     * becoming ridiculously small during transient outages like a
     * network partition.
     */
    double const RTCP_MIN_TIME = 5.;
    /*
     * Fraction of the RTCP bandwidth to be shared among active
     * senders.  (This fraction was chosen so that in a typical
     * session with one or two active senders, the computed report
     * time would be roughly equal to the minimum report time so that
     * we don't unnecessarily slow down receiver reports.) The
     * receiver fraction must be 1 - the sender fraction.
     */
    double const RTCP_SENDER_BW_FRACTION = 0.25;
    double const RTCP_RCVR_BW_FRACTION = (1-RTCP_SENDER_BW_FRACTION);
    /*
     * Gain (smoothing constant) for the low-pass filter that
```

```
     * estimates the average RTCP packet size (see Cadzow reference).
     */
    double const RTCP_SIZE_GAIN = (1./16.);

    double t;                   /* interval */
    double rtcp_min_time = RTCP_MIN_TIME;
    int n;                      /* no. of members for computation */

    /*
     * Very first call at application start-up uses half the min
     * delay for quicker notification while still allowing some time
     * before reporting for randomization and to learn about other
     * sources so the report interval will converge to the correct
     * interval more quickly.  The average RTCP size is initialized
     * to 128 octets which is conservative (it assumes everyone else
     * is generating SRs instead of RRs: 20 IP + 8 UDP + 52 SR + 48
     * SDES CNAME).
     */
    if (initial) {
        rtcp_min_time /= 2;
        *avg_rtcp_size = 128;
    }

    /*
     * If there were active senders, give them at least a minimum
     * share of the RTCP bandwidth.  Otherwise all participants share
     * the RTCP bandwidth equally.
     */
    n = members;
    if (senders > 0 && senders < members * RTCP_SENDER_BW_FRACTION) {
        if (we_sent) {
            rtcp_bw *= RTCP_SENDER_BW_FRACTION;
            n = senders;
        } else {
            rtcp_bw *= RTCP_RCVR_BW_FRACTION;
            n -= senders;
        }
    }

    /*
     * Update the average size estimate by the size of the report
     * packet we just sent.
     */
    *avg_rtcp_size += (packet_size - *avg_rtcp_size)*RTCP_SIZE_GAIN;

    /*
     * The effective number of sites times the average packet size is
     * the total number of octets sent when each site sends a report.
```

```
         * Dividing this by the effective bandwidth gives the time
         * interval over which those packets must be sent in order to
         * meet the bandwidth target, with a minimum enforced.  In that
         * time interval we send one report so this time is also our
         * average time between reports.
         */
        t = (*avg_rtcp_size) * n / rtcp_bw;
        if (t < rtcp_min_time) t = rtcp_min_time;

        /*
         * To avoid traffic bursts from unintended synchronization with
         * other sites, we then pick our actual next report interval as a
         * random number uniformly distributed between 0.5*t and 1.5*t.
         */
        return t * (drand48() + 0.5);
    }
```

A.8 Estimating the Interarrival Jitter

The code fragments below implement the algorithm given in Section 6.3.1 for calculating an estimate of the statistical variance of the RTP data interarrival time to be inserted in the interarrival jitter field of reception reports. The inputs are r->ts , the timestamp from the incoming packet, and arrival , the current time in the same units. Here s points to state for the source; s->transit holds the relative transit time for the previous packet, and s->jitter holds the estimated jitter. The jitter field of the reception report is measured in timestamp units and expressed as an unsigned integer, but the jitter estimate is kept in a floating point. As each data packet arrives, the jitter estimate is updated:

```
int transit = arrival - r->ts;
int d = transit - s->transit;
s->transit = transit;
if (d < 0) d = -d;
s->jitter += (1./16.) * ((double)d - s->jitter);
```

When a reception report block (to which rr points) is generated for this member, the current jitter estimate is returned:

```
rr->jitter = (u_int32) s->jitter;
```

Alternatively, the jitter estimate can be kept as an integer, but scaled to reduce round-off error. The calculation is the same except for the last line:

```
s->jitter += d - ((s->jitter + 8) >> 4);
```

In this case, the estimate is sampled for the reception report as:

```
rr->jitter = s->jitter >> 4;
```

B. Security Considerations

RTP suffers from the same security liabilities as the underlying protocols. For example, an impostor can fake source or destination network addresses, or change the header or payload. Within RTCP, the CNAME and NAME information may be used to impersonate another participant. In addition, RTP may be sent via IP multicast, which provides no direct means for a sender to know all the receivers of the data sent and therefore no measure of privacy. Rightly or not, users may be more sensitive to privacy concerns with audio and video communication than they have been with more traditional forms of network communication [24]. Therefore, the use of security mechanisms with RTP is important. These mechanisms are discussed in Section 9.

RTP-level translators or mixers may be used to allow RTP traffic to reach hosts behind firewalls. Appropriate firewall security principles and practices, which are beyond the scope of this document, should be followed in the design and installation of these devices and in the admission of RTP applications for use behind the firewall.

C. Authors' Addresses

Henning Schulzrinne
GMD Fokus
Hardenbergplatz 2
D-10623 Berlin
Germany

EMail: schulzrinne@fokus.gmd.de

Stephen L. Casner
Precept Software, Inc.
21580 Stevens Creek Boulevard, Suite 207
Cupertino, CA 95014
United States

EMail: casner@precept.com

Ron Frederick
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
United States

EMail: frederic@parc.xerox.com

Van Jacobson
MS 46a-1121
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
United States

EMail: van@ee.lbl.gov

Acknowledgments

This memorandum is based on discussions within the IETF Audio/Video Transport working group chaired by Stephen Casner. The current protocol has its origins in the Network Voice Protocol and the Packet Video Protocol (Danny Cohen and Randy Cole) and the protocol implemented by the vat application (Van Jacobson and Steve McCanne). Christian Huitema provided ideas for the random identifier generator.

D. Bibliography

[1] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in SIGCOMM Symposium on Communications Architectures and Protocols , (Philadelphia, Pennsylvania), pp. 200--208, IEEE, Sept. 1990. Computer Communications Review, Vol. 20(4), Sept. 1990.

[2] H. Schulzrinne, "Issues in designing a transport protocol for audio and video conferences and other multiparticipant real-time applications", Work in Progress.

[3] D. E. Comer, Internetworking with TCP/IP , vol. 1. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[4] Postel, J., "Internet Protocol", STD 5, RFC 791, USC/Information Sciences Institute, September 1981.

[5] Mills, D., "Network Time Protocol Version 3", RFC 1305, UDEL, March 1992.

[6] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1700, USC/Information Sciences Institute, October 1994.

[7] Eastlake, D., Crocker, S., and J. Schiller, "Randomness Recommendations for Security", RFC 1750, DEC, Cybercash, MIT, December 1994.

[8] J.-C. Bolot, T. Turletti, and I. Wakeman, "Scalable feedback control for multicast video distribution in the internet," in SIGCOMM Symposium on Communications Architectures and Protocols , (London, England), pp. 58--67, ACM, Aug. 1994.

[9] I. Busse, B. Deffner, and H. Schulzrinne, "Dynamic QoS control of multimedia applications based on RTP," Computer Communications , Jan. 1996.

[10] S. Floyd and V. Jacobson, "The synchronization of periodic routing messages," in SIGCOMM Symposium on Communications Architectures and Protocols (D. P. Sidhu, ed.), (San Francisco, California), pp. 33--44, ACM, Sept. 1993. also in [25].

[11] J. A. Cadzow, Foundations of digital signal processing and data analysis New York, New York: Macmillan, 1987.

[12] International Standards Organization, "ISO/IEC DIS 10646-1:1993 information technology -- universal multiple-octet coded character set (UCS) -- part I: Architecture and basic multilingual plane," 1993.

[13] The Unicode Consortium, The Unicode Standard New York, New York: Addison-Wesley, 1991.

[14] Mockapetris, P., "Domain Names - Concepts and Facilities", STD 13, RFC 1034, USC/Information Sciences Institute, November 1987.

[15] Mockapetris, P., "Domain Names - Implementation and Specification", STD 13, RFC 1035, USC/Information Sciences Institute, November 1987.

[16] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, Internet Engineering Task Force, October 1989.

[17] Rekhter, Y., Moskowitz, R., Karrenberg, D., and G. de Groot, "Address Allocation for Private Internets", RFC 1597, T.J. Watson Research Center, IBM Corp., Chrysler Corp., RIPE NCC, March 1994.

[18] Lear, E., Fair, E., Crocker, D., and T. Kessler, "Network 10 Considered Harmful (Some Practices Shouldn't be Codified)", RFC 1627, Silicon Graphics, Inc., Apple Computer, Inc., Silicon Graphics, Inc., July 1994.

[19] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.

[20] W. Feller, An Introduction to Probability Theory and its Applications, Volume 1 , vol. 1. New York, New York: John Wiley and Sons, third ed., 1968.

[21] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, TIS, IAB IRTF PSRG, IETF PEM WG, February 1993.

[22] V. L. Voydock and S. T. Kent, "Security mechanisms in high-level network protocols," ACM Computing Surveys , vol. 15, pp. 135--171, June 1983.

[23] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.

[24] S. Stubblebine, "Security services for multimedia conferencing," in 16th National Computer Security Conference , (Baltimore, Maryland), pp. 391--395, Sept. 1993.

[25] S. Floyd and V. Jacobson, "The synchronization of periodic routing messages," IEEE/ACM Transactions on Networking , vol. 2, pp. 122-136, April 1994.

Network Working Group
Request for Comments: 2326
Category: Standards Track

H. Schulzrinne
Columbia U.
A. Rao
Netscape
R. Lanphier
RealNetworks
April 1998

Real Time Streaming Protocol (RTSP)

Status of this Memo

Copyright Notice

Abstract

The Real Time Streaming Protocol, or RTSP, is an application-level protocol for control over the delivery of data with real-time properties. RTSP provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video. Sources of data can include both live data feeds and stored clips. This protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and TCP, and provide a means for choosing delivery mechanisms based upon RTP (RFC 1889).

Table of Contents

1 Introduction

1.1 Purpose

The Real-Time Streaming Protocol (RTSP) establishes and controls either a single or several time-synchronized streams of continuous media such as audio and video. It does not typically deliver the continuous streams itself, although interleaving of the continuous media stream with the control stream is possible (see Section 10.12). In other words, RTSP acts as a "network remote control" for multimedia servers.

The set of streams to be controlled is defined by a presentation description. This memorandum does not define a format for a presentation description.

There is no notion of an RTSP connection; instead, a server maintains a session labeled by an identifier. An RTSP session is in no way tied to a transport-level connection such as a TCP connection. During an RTSP session, an RTSP client may open and close many reliable transport connections to the server to issue RTSP requests. Alternatively, it may use a connectionless transport protocol such as UDP.

The streams controlled by RTSP may use RTP [1], but the operation of RTSP does not depend on the transport mechanism used to carry continuous media. The protocol is intentionally similar in syntax and operation to HTTP/1.1 [2] so that extension mechanisms to HTTP can in most cases also be added to RTSP. However, RTSP differs in a number of important aspects from HTTP:

* RTSP introduces a number of new methods and has a different protocol identifier.
* An RTSP server needs to maintain state by default in almost all cases, as opposed to the stateless nature of HTTP.
* Both an RTSP server and client can issue requests.
* Data is carried out-of-band by a different protocol. (There is an exception to this.)
* RTSP is defined to use ISO 10646 (UTF-8) rather than ISO 8859-1, consistent with current HTML internationalization efforts [3].
* The Request-URI always contains the absolute URI. Because of backward compatibility with a historical blunder, HTTP/1.1 [2] carries only the absolute path in the request and puts the host name in a separate header field.

This makes "virtual hosting" easier, where a single host with one IP address hosts several document trees.

The protocol supports the following operations:

Retrieval of media from media server:
: The client can request a presentation description via HTTP or some other method. If the presentation is being multicast, the presentation description contains the multicast addresses and ports to be used for the continuous media. If the presentation is to be sent only to the client via unicast, the client provides the destination for security reasons.

Invitation of a media server to a conference:
: A media server can be "invited" to join an existing conference, either to play back media into the presentation or to record all or a subset of the media in a presentation. This mode is useful for distributed teaching applications. Several parties in the conference may take turns "pushing the remote control buttons."

Addition of media to an existing presentation:
: Particularly for live presentations, it is useful if the server can tell the client about additional media becoming available.

RTSP requests may be handled by proxies, tunnels and caches as in HTTP/1.1 [2].

1.2 Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [4].

1.3 Terminology

Some of the terminology has been adopted from HTTP/1.1 [2]. Terms not listed here are defined as in HTTP/1.1.

Aggregate control:
: The control of the multiple streams using a single timeline by the server. For audio/video feeds, this means that the client may issue a single play or pause message to control both the audio and video feeds.

Conference:
: a multiparty, multimedia presentation, where "multi" implies greater than or equal to one.

Client:
The client requests continuous media data from the media server.

Connection:
A transport layer virtual circuit established between two programs for the purpose of communication.

Container file:
A file which may contain multiple media streams which often comprise a presentation when played together. RTSP servers may offer aggregate control on these files, though the concept of a container file is not embedded in the protocol.

Continuous media:
Data where there is a timing relationship between source and sink; that is, the sink must reproduce the timing relationship that existed at the source. The most common examples of continuous media are audio and motion video. Continuous media can be real-time (interactive), where there is a "tight" timing relationship between source and sink, or streaming (playback), where the relationship is less strict.

Entity:
The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body, as described in Section 8.

Media initialization:
Datatype/codec specific initialization. This includes such things as clockrates, color tables, etc. Any transport-independent information which is required by a client for playback of a media stream occurs in the media initialization phase of stream setup.

Media parameter:
Parameter specific to a media type that may be changed before or during stream playback.

Media server:
The server providing playback or recording services for one or more media streams. Different media streams within a presentation may originate from different media servers. A media server may reside on the same or a different host as the web server the presentation is invoked from.

Media server indirection:
Redirection of a media client to a different media server.

(Media) stream:
A single media instance, e.g., an audio stream or a video stream as well as a single whiteboard or shared application group. When using RTP, a stream consists of all RTP and RTCP packets created by a source within an RTP session. This is equivalent to the definition of a DSM-CC stream([5]).

Message:
The basic unit of RTSP communication, consisting of a structured sequence of octets matching the syntax defined in Section 15 and transmitted via a connection or a connectionless protocol.

Participant:
Member of a conference. A participant may be a machine, e.g., a media record or playback server.

Presentation:
A set of one or more streams presented to the client as a complete media feed, using a presentation description as defined below. In most cases in the RTSP context, this implies aggregate control of those streams, but does not have to.

Presentation description:
A presentation description contains information about one or more media streams within a presentation, such as the set of encodings, network addresses and information about the content. Other IETF protocols such as SDP (RFC 2327 [6]) use the term "session" for a live presentation. The presentation description may take several different formats, including but not limited to the session description format SDP.

Response:
An RTSP response. If an HTTP response is meant, that is indicated explicitly.

Request:
An RTSP request. If an HTTP request is meant, that is indicated explicitly.

RTSP session:
A complete RTSP "transaction", e.g., the viewing of a movie. A session typically consists of a client setting up a transport mechanism for the continuous media stream (SETUP), starting the stream with PLAY or RECORD, and closing the

stream with TEARDOWN.

Transport initialization:
The negotiation of transport information (e.g., port numbers, transport protocols) between the client and the server.

1.4 Protocol Properties

RTSP has the following properties:

Extendable:
New methods and parameters can be easily added to RTSP.

Easy to parse:
RTSP can be parsed by standard HTTP or MIME parsers.

Secure:
RTSP re-uses web security mechanisms. All HTTP authentication mechanisms such as basic (RFC 2068 [2, Section 11.1]) and digest authentication (RFC 2069 [8]) are directly applicable. One may also reuse transport or network layer security mechanisms.

Transport-independent:
RTSP may use either an unreliable datagram protocol (UDP) (RFC 768 [9]), a reliable datagram protocol (RDP, RFC 1151, not widely used [10]) or a reliable stream protocol such as TCP (RFC 793 [11]) as it implements application-level reliability.

Multi-server capable:
Each media stream within a presentation can reside on a different server. The client automatically establishes several concurrent control sessions with the different media servers. Media synchronization is performed at the transport level.

Control of recording devices:
The protocol can control both recording and playback devices, as well as devices that can alternate between the two modes ("VCR").

Separation of stream control and conference initiation:
Stream control is divorced from inviting a media server to a conference. The only requirement is that the conference initiation protocol either provides or can be used to create a unique conference identifier. In particular, SIP [12] or H.323 [13] may be used to invite a server to a conference.

Suitable for professional applications:
RTSP supports frame-level accuracy through SMPTE time stamps to allow remote digital editing.

Presentation description neutral:
The protocol does not impose a particular presentation description or metafile format and can convey the type of format to be used. However, the presentation description must contain at least one RTSP URI.

Proxy and firewall friendly:
The protocol should be readily handled by both application and transport-layer (SOCKS [14]) firewalls. A firewall may need to understand the SETUP method to open a "hole" for the UDP media stream.

HTTP-friendly:
Where sensible, RTSP reuses HTTP concepts, so that the existing infrastructure can be reused. This infrastructure includes PICS (Platform for Internet Content Selection [15,16]) for associating labels with content. However, RTSP does not just add methods to HTTP since the controlling continuous media requires server state in most cases.

Appropriate server control:
If a client can start a stream, it must be able to stop a stream. Servers should not start streaming to clients in such a way that clients cannot stop the stream.

Transport negotiation:
The client can negotiate the transport method prior to actually needing to process a continuous media stream.

Capability negotiation:
If basic features are disabled, there must be some clean mechanism for the client to determine which methods are not going to be implemented. This allows clients to present the appropriate user interface. For example, if seeking is not allowed, the user interface must be able to disallow moving a sliding position indicator.

An earlier requirement in RTSP was multi-client capability. However, it was determined that a better approach was to make sure that the protocol is easily extensible to the multi-client scenario. Stream identifiers can be used by several control streams, so that "passing the remote" would be possible. The protocol would not address how several clients negotiate access; this is left to either a "social protocol" or some other floor

control mechanism.

1.5 Extending RTSP

Since not all media servers have the same functionality, media servers by necessity will support different sets of requests. For example:

* A server may only be capable of playback thus has no need to support the RECORD request.
* A server may not be capable of seeking (absolute positioning) if it is to support live events only.
* Some servers may not support setting stream parameters and thus not support GET_PARAMETER and SET_PARAMETER.

A server SHOULD implement all header fields described in Section 12.

It is up to the creators of presentation descriptions not to ask the impossible of a server. This situation is similar in HTTP/1.1 [2], where the methods described in [H19.6] are not likely to be supported across all servers.

RTSP can be extended in three ways, listed here in order of the magnitude of changes supported:

* Existing methods can be extended with new parameters, as long as these parameters can be safely ignored by the recipient. (This is equivalent to adding new parameters to an HTML tag.) If the client needs negative acknowledgement when a method extension is not supported, a tag corresponding to the extension may be added in the Require: field (see Section 12.32).
* New methods can be added. If the recipient of the message does not understand the request, it responds with error code 501 (Not implemented) and the sender should not attempt to use this method again. A client may also use the OPTIONS method to inquire about methods supported by the server. The server SHOULD list the methods it supports using the Public response header.
* A new version of the protocol can be defined, allowing almost all aspects (except the position of the protocol version number) to change.

1.6 Overall Operation

Each presentation and media stream may be identified by an RTSP URL. The overall presentation and the properties of the media the presentation is made up of are defined by a presentation description file, the format of which is outside the scope of this specification. The presentation description file may be obtained by the client using

HTTP or other means such as email and may not necessarily be stored on the media server.

For the purposes of this specification, a presentation description is assumed to describe one or more presentations, each of which maintains a common time axis. For simplicity of exposition and without loss of generality, it is assumed that the presentation description contains exactly one such presentation. A presentation may contain several media streams.

The presentation description file contains a description of the media streams making up the presentation, including their encodings, language, and other parameters that enable the client to choose the most appropriate combination of media. In this presentation description, each media stream that is individually controllable by RTSP is identified by an RTSP URL, which points to the media server handling that particular media stream and names the stream stored on that server. Several media streams can be located on different servers; for example, audio and video streams can be split across servers for load sharing. The description also enumerates which transport methods the server is capable of.

Besides the media parameters, the network destination address and port need to be determined. Several modes of operation can be distinguished:

Unicast:
: The media is transmitted to the source of the RTSP request, with the port number chosen by the client. Alternatively, the media is transmitted on the same reliable stream as RTSP.

Multicast, server chooses address:
: The media server picks the multicast address and port. This is the typical case for a live or near-media-on-demand transmission.

Multicast, client chooses address:
: If the server is to participate in an existing multicast conference, the multicast address, port and encryption key are given by the conference description, established by means outside the scope of this specification.

1.7 RTSP States

RTSP controls a stream which may be sent via a separate protocol, independent of the control channel. For example, RTSP control may occur on a TCP connection while the data flows via UDP. Thus, data delivery continues even if no RTSP requests are received by the media

server. Also, during its lifetime, a single media stream may be controlled by RTSP requests issued sequentially on different TCP connections. Therefore, the server needs to maintain "session state" to be able to correlate RTSP requests with a stream. The state transitions are described in Section A.

Many methods in RTSP do not contribute to state. However, the following play a central role in defining the allocation and usage of stream resources on the server: SETUP, PLAY, RECORD, PAUSE, and TEARDOWN.

SETUP:
Causes the server to allocate resources for a stream and start an RTSP session.

PLAY and RECORD:
Starts data transmission on a stream allocated via SETUP.

PAUSE:
Temporarily halts a stream without freeing server resources.

TEARDOWN:
Frees resources associated with the stream. The RTSP session ceases to exist on the server.

RTSP methods that contribute to state use the Session header field (Section 12.37) to identify the RTSP session whose state is being manipulated. The server generates session identifiers in response to SETUP requests (Section 10.4).

1.8 Relationship with Other Protocols

RTSP has some overlap in functionality with HTTP. It also may interact with HTTP in that the initial contact with streaming content is often to be made through a web page. The current protocol specification aims to allow different hand-off points between a web server and the media server implementing RTSP. For example, the presentation description can be retrieved using HTTP or RTSP, which reduces roundtrips in web-browser-based scenarios, yet also allows for standalone RTSP servers and clients which do not rely on HTTP at all.

However, RTSP differs fundamentally from HTTP in that data delivery takes place out-of-band in a different protocol. HTTP is an asymmetric protocol where the client issues requests and the server responds. In RTSP, both the media client and media server can issue requests. RTSP requests are also not stateless; they may set parameters and continue to control a media stream long after the

request has been acknowledged.

Re-using HTTP functionality has advantages in at least two areas, namely security and proxies. The requirements are very similar, so having the ability to adopt HTTP work on caches, proxies and authentication is valuable.

While most real-time media will use RTP as a transport protocol, RTSP is not tied to RTP.

RTSP assumes the existence of a presentation description format that can express both static and temporal properties of a presentation containing several media streams.

2 Notational Conventions

Since many of the definitions and syntax are identical to HTTP/1.1, this specification only points to the section where they are defined rather than copying it. For brevity, [HX.Y] is to be taken to refer to Section X.Y of the current HTTP/1.1 specification (RFC 2068 [2]).

All the mechanisms specified in this document are described in both prose and an augmented Backus-Naur form (BNF) similar to that used in [H2.1]. It is described in detail in RFC 2234 [17], with the difference that this RTSP specification maintains the "1#" notation for comma-separated lists.

In this memo, we use indented and smaller-type paragraphs to provide background and motivation. This is intended to give readers who were not involved with the formulation of the specification an understanding of why things are the way that they are in RTSP.

3 Protocol Parameters

3.1 RTSP Version

[H3.1] applies, with HTTP replaced by RTSP.

3.2 RTSP URL

The "rtsp" and "rtspu" schemes are used to refer to network resources via the RTSP protocol. This section defines the scheme-specific syntax and semantics for RTSP URLs.

```
rtsp_URL  =   ( "rtsp:" | "rtspu:" )
              "//" host [ ":" port ] [ abs_path ]
host      =   <A legal Internet host domain name of IP address
              (in dotted decimal form), as defined by Section 2.1
```

```
                of RFC 1123 \cite{rfc1123}>
port        =   *DIGIT
```

abs_path is defined in [H3.2.1].

Note that fragment and query identifiers do not have a well-defined meaning at this time, with the interpretation left to the RTSP server.

The scheme rtsp requires that commands are issued via a reliable protocol (within the Internet, TCP), while the scheme rtspu identifies an unreliable protocol (within the Internet, UDP).

If the port is empty or not given, port 554 is assumed. The semantics are that the identified resource can be controlled by RTSP at the server listening for TCP (scheme "rtsp") connections or UDP (scheme "rtspu") packets on that port of host, and the Request-URI for the resource is rtsp_URL.

The use of IP addresses in URLs SHOULD be avoided whenever possible (see RFC 1924 [19]).

A presentation or a stream is identified by a textual media identifier, using the character set and escape conventions [H3.2] of URLs (RFC 1738 [20]). URLs may refer to a stream or an aggregate of streams, i.e., a presentation. Accordingly, requests described in Section 10 can apply to either the whole presentation or an individual stream within the presentation. Note that some request methods can only be applied to streams, not presentations and vice versa.

For example, the RTSP URL:

```
rtsp://media.example.com:554/twister/audiotrack
```

identifies the audio stream within the presentation "twister", which can be controlled via RTSP requests issued over a TCP connection to port 554 of host media.example.com.

Also, the RTSP URL:

```
rtsp://media.example.com:554/twister
```

identifies the presentation "twister", which may be composed of audio and video streams.

This does not imply a standard way to reference streams in URLs. The presentation description defines the hierarchical relationships in the presentation and the URLs for the individual streams. A presentation description may name a stream "a.mov" and the whole presentation "b.mov".

The path components of the RTSP URL are opaque to the client and do not imply any particular file system structure for the server.

This decoupling also allows presentation descriptions to be used with non-RTSP media control protocols simply by replacing the scheme in the URL.

3.3 Conference Identifiers

Conference identifiers are opaque to RTSP and are encoded using standard URI encoding methods (i.e., LWS is escaped with %). They can contain any octet value. The conference identifier MUST be globally unique. For H.323, the conferenceID value is to be used.

```
conference-id =   1*xchar
```

Conference identifiers are used to allow RTSP sessions to obtain parameters from multimedia conferences the media server is participating in. These conferences are created by protocols outside the scope of this specification, e.g., H.323 [13] or SIP [12]. Instead of the RTSP client explicitly providing transport information, for example, it asks the media server to use the values in the conference description instead.

3.4 Session Identifiers

Session identifiers are opaque strings of arbitrary length. Linear white space must be URL-escaped. A session identifier MUST be chosen randomly and MUST be at least eight octets long to make guessing it more difficult. (See Section 16.)

```
session-id   =   1*( ALPHA | DIGIT | safe )
```

3.5 SMPTE Relative Timestamps

A SMPTE relative timestamp expresses time relative to the start of the clip. Relative timestamps are expressed as SMPTE time codes for frame-level access accuracy. The time code has the format hours:minutes:seconds:frames.subframes, with the origin at the start of the clip. The default smpte format is "SMPTE 30 drop" format, with frame rate is 29.97 frames per second. Other SMPTE codes MAY be supported (such as "SMPTE 25") through the use of alternative use of "smpte time". For the "frames" field in the time value can assume the values 0 through 29. The difference between 30 and 29.97 frames per second is handled by dropping the first two frame indices (values 00 and 01) of every minute, except every tenth minute. If the frame value is zero, it may be omitted. Subframes are measured in one-hundredth of a frame.

```
smpte-range   =   smpte-type "=" smpte-time "-" [ smpte-time ]
smpte-type    =   "smpte" | "smpte-30-drop" | "smpte-25"
                                ; other timecodes may be added
smpte-time    =   1*2DIGIT ":" 1*2DIGIT ":" 1*2DIGIT [ ":" 1*2DIGIT ]
                     [ "." 1*2DIGIT ]
```

Examples:

```
smpte=10:12:33:20-
smpte=10:07:33-
smpte=10:07:00-10:07:33:05.01
smpte-25=10:07:00-10:07:33:05.01
```

3.6 Normal Play Time

Normal play time (NPT) indicates the stream absolute position relative to the beginning of the presentation. The timestamp consists of a decimal fraction. The part left of the decimal may be expressed in either seconds or hours, minutes, and seconds. The part right of the decimal point measures fractions of a second.

The beginning of a presentation corresponds to 0.0 seconds. Negative values are not defined. The special constant now is defined as the current instant of a live event. It may be used only for live events.

NPT is defined as in DSM-CC: "Intuitively, NPT is the clock the viewer associates with a program. It is often digitally displayed on a VCR. NPT advances normally when in normal play mode (scale = 1), advances at a faster rate when in fast scan forward (high positive scale ratio), decrements when in scan reverse (high negative scale ratio) and is fixed in pause mode. NPT is (logically) equivalent to SMPTE time codes." [5]

```
npt-range     =   ( npt-time "-" [ npt-time ] ) | ( "-" npt-time )
npt-time      =   "now" | npt-sec | npt-hhmmss
npt-sec       =   1*DIGIT [ "." *DIGIT ]
npt-hhmmss    =   npt-hh ":" npt-mm ":" npt-ss [ "." *DIGIT ]
npt-hh        =   1*DIGIT     ; any positive number
npt-mm        =   1*2DIGIT    ; 0-59
npt-ss        =   1*2DIGIT    ; 0-59
```

Examples:

```
npt=123.45-125
npt=12:05:35.3-
npt=now-
```

The syntax conforms to ISO 8601. The npt-sec notation is optimized for automatic generation, the ntp-hhmmss notation for consumption by human readers. The "now" constant allows clients to request to

receive the live feed rather than the stored or time-delayed version. This is needed since neither absolute time nor zero time are appropriate for this case.

3.7 Absolute Time

Absolute time is expressed as ISO 8601 timestamps, using UTC (GMT). Fractions of a second may be indicated.

```
utc-range     =   "clock" "=" utc-time "-" [ utc-time ]
utc-time      =   utc-date "T" utc-time "Z"
utc-date      =   8DIGIT                      ; < YYYYMMDD >
utc-time      =   6DIGIT [ "." fraction ]     ; < HHMMSS.fraction >
```

Example for November 8, 1996 at 14h37 and 20 and a quarter seconds UTC:

```
19961108T143720.25Z
```

3.8 Option Tags

Option tags are unique identifiers used to designate new options in RTSP. These tags are used in Require (Section 12.32) and Proxy-Require (Section 12.27) header fields.

Syntax:

```
option-tag    =   1*xchar
```

The creator of a new RTSP option should either prefix the option with a reverse domain name (e.g., "com.foo.mynewfeature" is an apt name for a feature whose inventor can be reached at "foo.com"), or register the new option with the Internet Assigned Numbers Authority (IANA).

3.8.1 Registering New Option Tags with IANA

When registering a new RTSP option, the following information should be provided:

* Name and description of option. The name may be of any length, but SHOULD be no more than twenty characters long. The name MUST not contain any spaces, control characters or periods.
* Indication of who has change control over the option (for example, IETF, ISO, ITU-T, other international standardization bodies, a consortium or a particular company or group of companies);

* A reference to a further description, if available, for example (in order of preference) an RFC, a published paper, a patent filing, a technical report, documented source code or a computer manual;
* For proprietary options, contact information (postal and email address);

4 RTSP Message

RTSP is a text-based protocol and uses the ISO 10646 character set in UTF-8 encoding (RFC 2279 [21]). Lines are terminated by CRLF, but receivers should be prepared to also interpret CR and LF by themselves as line terminators.

Text-based protocols make it easier to add optional parameters in a self-describing manner. Since the number of parameters and the frequency of commands is low, processing efficiency is not a concern. Text-based protocols, if done carefully, also allow easy implementation of research prototypes in scripting languages such as Tcl, Visual Basic and Perl.

The 10646 character set avoids tricky character set switching, but is invisible to the application as long as US-ASCII is being used. This is also the encoding used for RTCP. ISO 8859-1 translates directly into Unicode with a high-order octet of zero. ISO 8859-1 characters with the most-significant bit set are represented as 1100001x 10xxxxxx. (See RFC 2279 [21])

RTSP messages can be carried over any lower-layer transport protocol that is 8-bit clean.

Requests contain methods, the object the method is operating upon and parameters to further describe the method. Methods are idempotent, unless otherwise noted. Methods are also designed to require little or no state maintenance at the media server.

4.1 Message Types

See [H4.1]

4.2 Message Headers

See [H4.2]

4.3 Message Body

See [H4.3]

4.4 Message Length

When a message body is included with a message, the length of that body is determined by one of the following (in order of precedence):

1. Any response message which MUST NOT include a message body (such as the 1xx, 204, and 304 responses) is always terminated by the first empty line after the header fields, regardless of the entity-header fields present in the message. (Note: An empty line consists of only CRLF.)

2. If a Content-Length header field (section 12.14) is present, its value in bytes represents the length of the message-body. If this header field is not present, a value of zero is assumed.

3. By the server closing the connection. (Closing the connection cannot be used to indicate the end of a request body, since that would leave no possibility for the server to send back a response.)

Note that RTSP does not (at present) support the HTTP/1.1 "chunked" transfer coding(see [H3.6]) and requires the presence of the Content-Length header field.

Given the moderate length of presentation descriptions returned, the server should always be able to determine its length, even if it is generated dynamically, making the chunked transfer encoding unnecessary. Even though Content-Length must be present if there is any entity body, the rules ensure reasonable behavior even if the length is not given explicitly.

5 General Header Fields

See [H4.5], except that Pragma, Transfer-Encoding and Upgrade headers are not defined:

```
general-header  =  Cache-Control      ; Section 12.8
                |  Connection         ; Section 12.10
                |  Date               ; Section 12.18
                |  Via                ; Section 12.43
```

6 Request

A request message from a client to a server or vice versa includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request       =       Request-Line            ; Section 6.1
                *(      general-header          ; Section 5
                |       request-header         ; Section 6.2
                |       entity-header )        ; Section 8.1
                        CRLF
                        [ message-body ]        ; Section 4.3
```

6.1 Request Line

```
Request-Line = Method SP Request-URI SP RTSP-Version CRLF

Method          =       "DESCRIBE"              ; Section 10.2
                |       "ANNOUNCE"              ; Section 10.3
                |       "GET_PARAMETER"         ; Section 10.8
                |       "OPTIONS"               ; Section 10.1
                |       "PAUSE"                 ; Section 10.6
                |       "PLAY"                  ; Section 10.5
                |       "RECORD"                ; Section 10.11
                |       "REDIRECT"              ; Section 10.10
                |       "SETUP"                 ; Section 10.4
                |       "SET_PARAMETER"         ; Section 10.9
                |       "TEARDOWN"              ; Section 10.7
                |       extension-method

extension-method = token

Request-URI = "*" | absolute_URI

RTSP-Version = "RTSP" "/" 1*DIGIT "." 1*DIGIT
```

6.2 Request Header Fields

```
request-header  =       Accept                  ; Section 12.1
                |       Accept-Encoding          ; Section 12.2
                |       Accept-Language          ; Section 12.3
                |       Authorization            ; Section 12.5
                |       From                     ; Section 12.20
                |       If-Modified-Since        ; Section 12.23
                |       Range                    ; Section 12.29
                |       Referer                  ; Section 12.30
                |       User-Agent               ; Section 12.41
```

Note that in contrast to HTTP/1.1 [2], RTSP requests always contain the absolute URL (that is, including the scheme, host and port) rather than just the absolute path.

HTTP/1.1 requires servers to understand the absolute URL, but clients are supposed to use the Host request header. This is purely needed for backward-compatibility with HTTP/1.0 servers, a consideration that does not apply to RTSP.

The asterisk "*" in the Request-URI means that the request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. One example would be:

```
OPTIONS * RTSP/1.0
```

7 Response

[H6] applies except that HTTP-Version is replaced by RTSP-Version. Also, RTSP defines additional status codes and does not define some HTTP codes. The valid response codes and the methods they can be used with are defined in Table 1.

After receiving and interpreting a request message, the recipient responds with an RTSP response message.

```
Response    =     Status-Line          ; Section 7.1
              *(  general-header       ; Section 5
              |   response-header      ; Section 7.1.2
              |   entity-header )      ; Section 8.1
                  CRLF
                  [ message-body ]     ; Section 4.3
```

7.1 Status-Line

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code, and the textual phrase associated with the status code, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
Status-Line =   RTSP-Version SP Status-Code SP Reason-Phrase CRLF
```

7.1.1 Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in Section 11. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

* 1xx: Informational - Request received, continuing process
* 2xx: Success - The action was successfully received, understood, and accepted
* 3xx: Redirection - Further action must be taken in order to complete the request
* 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
* 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for RTSP/1.0, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommended - they may be replaced by local equivalents without affecting the protocol. Note that RTSP adopts most HTTP/1.1 [2] status codes and adds RTSP-specific status codes starting at x50 to avoid conflicts with newly defined HTTP status codes.

```
   Status-Code  =      "100"       ; Continue
                |      "200"       ; OK
                |      "201"       ; Created
                |      "250"       ; Low on Storage Space
                |      "300"       ; Multiple Choices
                |      "301"       ; Moved Permanently
                |      "302"       ; Moved Temporarily
                |      "303"       ; See Other
                |      "304"       ; Not Modified
                |      "305"       ; Use Proxy
                |      "400"       ; Bad Request
                |      "401"       ; Unauthorized
                |      "402"       ; Payment Required
                |      "403"       ; Forbidden
                |      "404"       ; Not Found
                |      "405"       ; Method Not Allowed
                |      "406"       ; Not Acceptable
                |      "407"       ; Proxy Authentication Required
                |      "408"       ; Request Time-out
                |      "410"       ; Gone
                |      "411"       ; Length Required
                |      "412"       ; Precondition Failed
                |      "413"       ; Request Entity Too Large
                |      "414"       ; Request-URI Too Large
                |      "415"       ; Unsupported Media Type
                |      "451"       ; Parameter Not Understood
                |      "452"       ; Conference Not Found
                |      "453"       ; Not Enough Bandwidth
                |      "454"       ; Session Not Found
                |      "455"       ; Method Not Valid in This State
                |      "456"       ; Header Field Not Valid for Resource
                |      "457"       ; Invalid Range
                |      "458"       ; Parameter Is Read-Only
                |      "459"       ; Aggregate operation not allowed
                |      "460"       ; Only aggregate operation allowed
                |      "461"       ; Unsupported transport
                |      "462"       ; Destination unreachable
                |      "500"       ; Internal Server Error
                |      "501"       ; Not Implemented
                |      "502"       ; Bad Gateway
                |      "503"       ; Service Unavailable
                |      "504"       ; Gateway Time-out
                |      "505"       ; RTSP Version not supported
                |      "551"       ; Option not supported
                |     extension-code
```

extension-code = 3DIGIT

Reason-Phrase = *<TEXT, excluding CR, LF>

RTSP status codes are extensible. RTSP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications MUST understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response MUST NOT be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents SHOULD present to the user the entity returned with the response, since that entity is likely to include human-readable information which will explain the unusual status.

Code	reason	
100	Continue	all
200	OK	all
201	Created	RECORD
250	Low on Storage Space	RECORD
300	Multiple Choices	all
301	Moved Permanently	all
302	Moved Temporarily	all
303	See Other	all
305	Use Proxy	all

400	Bad Request	all
401	Unauthorized	all
402	Payment Required	all
403	Forbidden	all
404	Not Found	all
405	Method Not Allowed	all
406	Not Acceptable	all
407	Proxy Authentication Required	all
408	Request Timeout	all
410	Gone	all
411	Length Required	all
412	Precondition Failed	DESCRIBE, SETUP
413	Request Entity Too Large	all
414	Request-URI Too Long	all
415	Unsupported Media Type	all
451	Invalid parameter	SETUP
452	Illegal Conference Identifier	SETUP
453	Not Enough Bandwidth	SETUP
454	Session Not Found	all
455	Method Not Valid In This State	all
456	Header Field Not Valid	all
457	Invalid Range	PLAY
458	Parameter Is Read-Only	SET_PARAMETER
459	Aggregate Operation Not Allowed	all
460	Only Aggregate Operation Allowed	all
461	Unsupported Transport	all
462	Destination Unreachable	all
500	Internal Server Error	all
501	Not Implemented	all
502	Bad Gateway	all
503	Service Unavailable	all
504	Gateway Timeout	all
505	RTSP Version Not Supported	all
551	Option not support	all

Table 1: Status codes and their usage with RTSP methods

7.1.2 Response Header Fields

The response-header fields allow the request recipient to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

```
response-header  =    Location              ; Section 12.25
                 |    Proxy-Authenticate     ; Section 12.26
                 |    Public                 ; Section 12.28
                 |    Retry-After            ; Section 12.31
                 |    Server                 ; Section 12.36
                 |    Vary                   ; Section 12.42
                 |    WWW-Authenticate       ; Section 12.44
```

Response-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response-header fields if all parties in the communication recognize them to be response-header fields. Unrecognized header fields are treated as entity-header fields.

8 Entity

Request and Response messages MAY transfer an entity if not otherwise restricted by the request method or response status code. An entity consists of entity-header fields and an entity-body, although some responses will only include the entity-headers.

In this section, both sender and recipient refer to either the client or the server, depending on who sends and who receives the entity.

8.1 Entity Header Fields

Entity-header fields define optional metainformation about the entity-body or, if no body is present, about the resource identified by the request.

```
entity-header    =    Allow                 ; Section 12.4
                 |    Content-Base          ; Section 12.11
                 |    Content-Encoding      ; Section 12.12
                 |    Content-Language      ; Section 12.13
                 |    Content-Length        ; Section 12.14
                 |    Content-Location      ; Section 12.15
                 |    Content-Type          ; Section 12.16
                 |    Expires               ; Section 12.19
                 |    Last-Modified         ; Section 12.24
                 |    extension-header
extension-header =    message-header
```

The extension-header mechanism allows additional entity-header fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields SHOULD be ignored by the recipient and forwarded by proxies.

8.2 Entity Body

See [H7.2]

9 Connections

RTSP requests can be transmitted in several different ways:

* persistent transport connections used for several request-response transactions;
* one connection per request/response transaction;
* connectionless mode.

The type of transport connection is defined by the RTSP URI (Section 3.2). For the scheme "rtsp", a persistent connection is assumed, while the scheme "rtspu" calls for RTSP requests to be sent without setting up a connection.

Unlike HTTP, RTSP allows the media server to send requests to the media client. However, this is only supported for persistent connections, as the media server otherwise has no reliable way of reaching the client. Also, this is the only way that requests from media server to client are likely to traverse firewalls.

9.1 Pipelining

A client that supports persistent connections or connectionless mode MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MUST send its responses to those requests in the same order that the requests were received.

9.2 Reliability and Acknowledgements

Requests are acknowledged by the receiver unless they are sent to a multicast group. If there is no acknowledgement, the sender may resend the same message after a timeout of one round-trip time (RTT). The round-trip time is estimated as in TCP (RFC 1123) [18], with an initial round-trip value of 500 ms. An implementation MAY cache the last RTT measurement as the initial value for future connections.

If a reliable transport protocol is used to carry RTSP, requests MUST NOT be retransmitted; the RTSP application MUST instead rely on the underlying transport to provide reliability.

If both the underlying reliable transport such as TCP and the RTSP application retransmit requests, it is possible that each packet loss results in two retransmissions. The receiver cannot typically take advantage of the application-layer retransmission since the

transport stack will not deliver the application-layer retransmission before the first attempt has reached the receiver. If the packet loss is caused by congestion, multiple retransmissions at different layers will exacerbate the congestion.

If RTSP is used over a small-RTT LAN, standard procedures for optimizing initial TCP round trip estimates, such as those used in T/TCP (RFC 1644) [22], can be beneficial.

The Timestamp header (Section 12.38) is used to avoid the retransmission ambiguity problem [23, p. 301] and obviates the need for Karn's algorithm.

Each request carries a sequence number in the CSeq header (Section 12.17), which is incremented by one for each distinct request transmitted. If a request is repeated because of lack of acknowledgement, the request MUST carry the original sequence number (i.e., the sequence number is not incremented).

Systems implementing RTSP MUST support carrying RTSP over TCP and MAY support UDP. The default port for the RTSP server is 554 for both UDP and TCP.

A number of RTSP packets destined for the same control end point may be packed into a single lower-layer PDU or encapsulated into a TCP stream. RTSP data MAY be interleaved with RTP and RTCP packets. Unlike HTTP, an RTSP message MUST contain a Content-Length header whenever that message contains a payload. Otherwise, an RTSP packet is terminated with an empty line immediately following the last message header.

10 Method Definitions

The method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive. New methods may be defined in the future. Method names may not start with a $ character (decimal 24) and must be a token. Methods are summarized in Table 2.

method	direction	object	requirement
DESCRIBE	C->S	P,S	recommended
ANNOUNCE	C->S, S->C	P,S	optional
GET_PARAMETER	C->S, S->C	P,S	optional
OPTIONS	C->S, S->C	P,S	required (S->C: optional)
PAUSE	C->S	P,S	recommended
PLAY	C->S	P,S	required
RECORD	C->S	P,S	optional
REDIRECT	S->C	P,S	optional
SETUP	C->S	S	required
SET_PARAMETER	C->S, S->C	P,S	optional
TEARDOWN	C->S	P,S	required

Table 2: Overview of RTSP methods, their direction, and what objects (P: presentation, S: stream) they operate on

Notes on Table 2: PAUSE is recommended, but not required in that a fully functional server can be built that does not support this method, for example, for live feeds. If a server does not support a particular method, it MUST return "501 Not Implemented" and a client SHOULD not try this method again for this server.

10.1 OPTIONS

The behavior is equivalent to that described in [H9.2]. An OPTIONS request may be issued at any time, e.g., if the client is about to try a nonstandard request. It does not influence server state.

Example:

```
C->S:  OPTIONS * RTSP/1.0
       CSeq: 1
       Require: implicit-play
       Proxy-Require: gzipped-messages

S->C:  RTSP/1.0 200 OK
       CSeq: 1
       Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE
```

Note that these are necessarily fictional features (one would hope that we would not purposefully overlook a truly useful feature just so that we could have a strong example in this section).

10.2 DESCRIBE

The DESCRIBE method retrieves the description of a presentation or media object identified by the request URL from a server. It may use the Accept header to specify the description formats that the client understands. The server responds with a description of the requested resource. The DESCRIBE reply-response pair constitutes the media initialization phase of RTSP.

Example:

```
C->S: DESCRIBE rtsp://server.example.com/fizzle/foo RTSP/1.0
      CSeq: 312
      Accept: application/sdp, application/rtsl, application/mheg

S->C: RTSP/1.0 200 OK
      CSeq: 312
      Date: 23 Jan 1997 15:35:06 GMT
      Content-Type: application/sdp
      Content-Length: 376

      v=0
      o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
      s=SDP Seminar
      i=A Seminar on the session description protocol
      u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
      e=mjh@isi.edu (Mark Handley)
      c=IN IP4 224.2.17.12/127
      t=2873397496 2873404696
      a=recvonly
      m=audio 3456 RTP/AVP 0
      m=video 2232 RTP/AVP 31
      m=whiteboard 32416 UDP WB
      a=orient:portrait
```

The DESCRIBE response MUST contain all media initialization information for the resource(s) that it describes. If a media client obtains a presentation description from a source other than DESCRIBE and that description contains a complete set of media initialization parameters, the client SHOULD use those parameters and not then request a description for the same media via RTSP.

Additionally, servers SHOULD NOT use the DESCRIBE response as a means of media indirection.

Clear ground rules need to be established so that clients have an unambiguous means of knowing when to request media initialization information via DESCRIBE, and when not to. By forcing a DESCRIBE

response to contain all media initialization for the set of streams that it describes, and discouraging use of DESCRIBE for media indirection, we avoid looping problems that might result from other approaches.

Media initialization is a requirement for any RTSP-based system, but the RTSP specification does not dictate that this must be done via the DESCRIBE method. There are three ways that an RTSP client may receive initialization information:

* via RTSP's DESCRIBE method;
* via some other protocol (HTTP, email attachment, etc.);
* via the command line or standard input (thus working as a browser helper application launched with an SDP file or other media initialization format).

In the interest of practical interoperability, it is highly recommended that minimal servers support the DESCRIBE method, and highly recommended that minimal clients support the ability to act as a "helper application" that accepts a media initialization file from standard input, command line, and/or other means that are appropriate to the operating environment of the client.

10.3 ANNOUNCE

The ANNOUNCE method serves two purposes:

When sent from client to server, ANNOUNCE posts the description of a presentation or media object identified by the request URL to a server. When sent from server to client, ANNOUNCE updates the session description in real-time.

If a new media stream is added to a presentation (e.g., during a live presentation), the whole presentation description should be sent again, rather than just the additional components, so that components can be deleted.

Example:

```
C->S: ANNOUNCE rtsp://server.example.com/fizzle/foo RTSP/1.0
      CSeq: 312
      Date: 23 Jan 1997 15:35:06 GMT
      Session: 47112344
      Content-Type: application/sdp
      Content-Length: 332

      v=0
      o=mhandley 2890844526 2890845468 IN IP4 126.16.64.4
```

```
        s=SDP Seminar
        i=A Seminar on the session description protocol
        u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
        e=mjh@isi.edu (Mark Handley)
        c=IN IP4 224.2.17.12/127
        t=2873397496 2873404696
        a=recvonly
        m=audio 3456 RTP/AVP 0
        m=video 2232 RTP/AVP 31

   S->C: RTSP/1.0 200 OK
        CSeq: 312
```

10.4 SETUP

The SETUP request for a URI specifies the transport mechanism to be used for the streamed media. A client can issue a SETUP request for a stream that is already playing to change transport parameters, which a server MAY allow. If it does not allow this, it MUST respond with error "455 Method Not Valid In This State". For the benefit of any intervening firewalls, a client must indicate the transport parameters even if it has no influence over these parameters, for example, where the server advertises a fixed multicast address.

> Since SETUP includes all transport initialization information, firewalls and other intermediate network devices (which need this information) are spared the more arduous task of parsing the DESCRIBE response, which has been reserved for media initialization.

The Transport header specifies the transport parameters acceptable to the client for data transmission; the response will contain the transport parameters selected by the server.

```
   C->S: SETUP rtsp://example.com/foo/bar/baz.rm RTSP/1.0
        CSeq: 302
        Transport: RTP/AVP;unicast;client_port=4588-4589

   S->C: RTSP/1.0 200 OK
        CSeq: 302
        Date: 23 Jan 1997 15:35:06 GMT
        Session: 47112344
        Transport: RTP/AVP;unicast;
          client_port=4588-4589;server_port=6256-6257
```

The server generates session identifiers in response to SETUP requests. If a SETUP request to a server includes a session identifier, the server MUST bundle this setup request into the

existing session or return error "459 Aggregate Operation Not Allowed" (see Section 11.3.10).

10.5 PLAY

The PLAY method tells the server to start sending data via the mechanism specified in SETUP. A client MUST NOT issue a PLAY request until any outstanding SETUP requests have been acknowledged as successful.

The PLAY request positions the normal play time to the beginning of the range specified and delivers stream data until the end of the range is reached. PLAY requests may be pipelined (queued); a server MUST queue PLAY requests to be executed in order. That is, a PLAY request arriving while a previous PLAY request is still active is delayed until the first has been completed.

This allows precise editing.

For example, regardless of how closely spaced the two PLAY requests in the example below arrive, the server will first play seconds 10 through 15, then, immediately following, seconds 20 to 25, and finally seconds 30 through the end.

```
C->S: PLAY rtsp://audio.example.com/audio RTSP/1.0
      CSeq: 835
      Session: 12345678
      Range: npt=10-15

C->S: PLAY rtsp://audio.example.com/audio RTSP/1.0
      CSeq: 836
      Session: 12345678
      Range: npt=20-25

C->S: PLAY rtsp://audio.example.com/audio RTSP/1.0
      CSeq: 837
      Session: 12345678
      Range: npt=30-
```

See the description of the PAUSE request for further examples.

A PLAY request without a Range header is legal. It starts playing a stream from the beginning unless the stream has been paused. If a stream has been paused via PAUSE, stream delivery resumes at the pause point. If a stream is playing, such a PLAY request causes no further action and can be used by the client to test server liveness.

The Range header may also contain a time parameter. This parameter specifies a time in UTC at which the playback should start. If the message is received after the specified time, playback is started immediately. The time parameter may be used to aid in synchronization of streams obtained from different sources.

For a on-demand stream, the server replies with the actual range that will be played back. This may differ from the requested range if alignment of the requested range to valid frame boundaries is required for the media source. If no range is specified in the request, the current position is returned in the reply. The unit of the range in the reply is the same as that in the request.

After playing the desired range, the presentation is automatically paused, as if a PAUSE request had been issued.

The following example plays the whole presentation starting at SMPTE time code 0:10:20 until the end of the clip. The playback is to start at 15:36 on 23 Jan 1997.

```
C->S: PLAY rtsp://audio.example.com/twister.en RTSP/1.0
      CSeq: 833
      Session: 12345678
      Range: smpte=0:10:20-;time=19970123T153600Z

S->C: RTSP/1.0 200 OK
      CSeq: 833
      Date: 23 Jan 1997 15:35:06 GMT
      Range: smpte=0:10:22-;time=19970123T153600Z
```

For playing back a recording of a live presentation, it may be desirable to use clock units:

```
C->S: PLAY rtsp://audio.example.com/meeting.en RTSP/1.0
      CSeq: 835
      Session: 12345678
      Range: clock=19961108T142300Z-19961108T143520Z

S->C: RTSP/1.0 200 OK
      CSeq: 835
      Date: 23 Jan 1997 15:35:06 GMT
```

A media server only supporting playback MUST support the npt format and MAY support the clock and smpte formats.

10.6 PAUSE

The PAUSE request causes the stream delivery to be interrupted (halted) temporarily. If the request URL names a stream, only playback and recording of that stream is halted. For example, for audio, this is equivalent to muting. If the request URL names a presentation or group of streams, delivery of all currently active streams within the presentation or group is halted. After resuming playback or recording, synchronization of the tracks MUST be maintained. Any server resources are kept, though servers MAY close the session and free resources after being paused for the duration specified with the timeout parameter of the Session header in the SETUP message.

Example:

```
C->S: PAUSE rtsp://example.com/fizzle/foo RTSP/1.0
      CSeq: 834
      Session: 12345678

S->C: RTSP/1.0 200 OK
      CSeq: 834
      Date: 23 Jan 1997 15:35:06 GMT
```

The PAUSE request may contain a Range header specifying when the stream or presentation is to be halted. We refer to this point as the "pause point". The header must contain exactly one value rather than a time range. The normal play time for the stream is set to the pause point. The pause request becomes effective the first time the server is encountering the time point specified in any of the currently pending PLAY requests. If the Range header specifies a time outside any currently pending PLAY requests, the error "457 Invalid Range" is returned. If a media unit (such as an audio or video frame) starts presentation at exactly the pause point, it is not played or recorded. If the Range header is missing, stream delivery is interrupted immediately on receipt of the message and the pause point is set to the current normal play time.

A PAUSE request discards all queued PLAY requests. However, the pause point in the media stream MUST be maintained. A subsequent PLAY request without Range header resumes from the pause point.

For example, if the server has play requests for ranges 10 to 15 and 20 to 29 pending and then receives a pause request for NPT 21, it would start playing the second range and stop at NPT 21. If the pause request is for NPT 12 and the server is playing at NPT 13 serving the first play request, the server stops immediately. If the pause request is for NPT 16, the server stops after completing the first

play request and discards the second play request.

As another example, if a server has received requests to play ranges 10 to 15 and then 13 to 20 (that is, overlapping ranges), the PAUSE request for NPT=14 would take effect while the server plays the first range, with the second PLAY request effectively being ignored, assuming the PAUSE request arrives before the server has started playing the second, overlapping range. Regardless of when the PAUSE request arrives, it sets the NPT to 14.

If the server has already sent data beyond the time specified in the Range header, a PLAY would still resume at that point in time, as it is assumed that the client has discarded data after that point. This ensures continuous pause/play cycling without gaps.

10.7 TEARDOWN

The TEARDOWN request stops the stream delivery for the given URI, freeing the resources associated with it. If the URI is the presentation URI for this presentation, any RTSP session identifier associated with the session is no longer valid. Unless all transport parameters are defined by the session description, a SETUP request has to be issued before the session can be played again.

Example:

```
  C->S: TEARDOWN rtsp://example.com/fizzle/foo RTSP/1.0
        CSeq: 892
        Session: 12345678
  S->C: RTSP/1.0 200 OK
        CSeq: 892
```

10.8 GET_PARAMETER

The GET_PARAMETER request retrieves the value of a parameter of a presentation or stream specified in the URI. The content of the reply and response is left to the implementation. GET_PARAMETER with no entity body may be used to test client or server liveness ("ping").

Example:

```
  S->C: GET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
        CSeq: 431
        Content-Type: text/parameters
        Session: 12345678
        Content-Length: 15

        packets_received
        jitter
```

```
C->S: RTSP/1.0 200 OK
      CSeq: 431
      Content-Length: 46
      Content-Type: text/parameters

      packets_received: 10
      jitter: 0.3838
```

The "text/parameters" section is only an example type for parameter. This method is intentionally loosely defined with the intention that the reply content and response content will be defined after further experimentation.

10.9 SET_PARAMETER

This method requests to set the value of a parameter for a presentation or stream specified by the URI.

A request SHOULD only contain a single parameter to allow the client to determine why a particular request failed. If the request contains several parameters, the server MUST only act on the request if all of the parameters can be set successfully. A server MUST allow a parameter to be set repeatedly to the same value, but it MAY disallow changing parameter values.

Note: transport parameters for the media stream MUST only be set with the SETUP command.

Restricting setting transport parameters to SETUP is for the benefit of firewalls.

The parameters are split in a fine-grained fashion so that there can be more meaningful error indications. However, it may make sense to allow the setting of several parameters if an atomic setting is desirable. Imagine device control where the client does not want the camera to pan unless it can also tilt to the right angle at the same time.

Example:

```
C->S: SET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
      CSeq: 421
      Content-length: 20
      Content-type: text/parameters

      barparam: barstuff

S->C: RTSP/1.0 451 Invalid Parameter
```

```
CSeq: 421
Content-length: 10
Content-type: text/parameters

barparam
```

The "text/parameters" section is only an example type for parameter. This method is intentionally loosely defined with the intention that the reply content and response content will be defined after further experimentation.

10.10 REDIRECT

A redirect request informs the client that it must connect to another server location. It contains the mandatory header Location, which indicates that the client should issue requests for that URL. It may contain the parameter Range, which indicates when the redirection takes effect. If the client wants to continue to send or receive media for this URI, the client MUST issue a TEARDOWN request for the current session and a SETUP for the new session at the designated host.

This example request redirects traffic for this URI to the new server at the given play time:

```
S->C: REDIRECT rtsp://example.com/fizzle/foo RTSP/1.0
      CSeq: 732
      Location: rtsp://bigserver.com:8001
      Range: clock=19960213T143205Z-
```

10.11 RECORD

This method initiates recording a range of media data according to the presentation description. The timestamp reflects start and end time (UTC). If no time range is given, use the start or end time provided in the presentation description. If the session has already started, commence recording immediately.

The server decides whether to store the recorded data under the request-URI or another URI. If the server does not use the request-URI, the response SHOULD be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

A media server supporting recording of live presentations MUST support the clock range format; the smpte format does not make sense.

In this example, the media server was previously invited to the conference indicated.

```
C->S: RECORD rtsp://example.com/meeting/audio.en RTSP/1.0
      CSeq: 954
      Session: 12345678
      Conference: 128.16.64.19/32492374
```

10.12 Embedded (Interleaved) Binary Data

Certain firewall designs and other circumstances may force a server to interleave RTSP methods and stream data. This interleaving should generally be avoided unless necessary since it complicates client and server operation and imposes additional overhead. Interleaved binary data SHOULD only be used if RTSP is carried over TCP.

Stream data such as RTP packets is encapsulated by an ASCII dollar sign (24 hexadecimal), followed by a one-byte channel identifier, followed by the length of the encapsulated binary data as a binary, two-byte integer in network byte order. The stream data follows immediately afterwards, without a CRLF, but including the upper-layer protocol headers. Each $ block contains exactly one upper-layer protocol data unit, e.g., one RTP packet.

The channel identifier is defined in the Transport header with the interleaved parameter(Section 12.39).

When the transport choice is RTP, RTCP messages are also interleaved by the server over the TCP connection. As a default, RTCP packets are sent on the first available channel higher than the RTP channel. The client MAY explicitly request RTCP packets on another channel. This is done by specifying two channels in the interleaved parameter of the Transport header(Section 12.39).

RTCP is needed for synchronization when two or more streams are interleaved in such a fashion. Also, this provides a convenient way to tunnel RTP/RTCP packets through the TCP control connection when required by the network configuration and transfer them onto UDP when possible.

```
C->S: SETUP rtsp://foo.com/bar.file RTSP/1.0
      CSeq: 2
      Transport: RTP/AVP/TCP;interleaved=0-1

S->C: RTSP/1.0 200 OK
      CSeq: 2
      Date: 05 Jun 1997 18:57:18 GMT
      Transport: RTP/AVP/TCP;interleaved=0-1
```

```
        Session: 12345678

   C->S: PLAY rtsp://foo.com/bar.file RTSP/1.0
        CSeq: 3
        Session: 12345678

   S->C: RTSP/1.0 200 OK
        CSeq: 3
        Session: 12345678
        Date: 05 Jun 1997 18:59:15 GMT
        RTP-Info: url=rtsp://foo.com/bar.file;
          seq=232433;rtptime=972948234

   S->C: $\000{2 byte length}{"length" bytes data, w/RTP header}
   S->C: $\000{2 byte length}{"length" bytes data, w/RTP header}
   S->C: $\001{2 byte length}{"length" bytes  RTCP packet}
```

11 Status Code Definitions

Where applicable, HTTP status [H10] codes are reused. Status codes that have the same meaning are not repeated here. See Table 1 for a listing of which status codes may be returned by which requests.

11.1 Success 2xx

11.1.1 250 Low on Storage Space

The server returns this warning after receiving a RECORD request that it may not be able to fulfill completely due to insufficient storage space. If possible, the server should use the Range header to indicate what time period it may still be able to record. Since other processes on the server may be consuming storage space simultaneously, a client should take this only as an estimate.

11.2 Redirection 3xx

See [H10.3].

Within RTSP, redirection may be used for load balancing or redirecting stream requests to a server topologically closer to the client. Mechanisms to determine topological proximity are beyond the scope of this specification.

11.3 Client Error 4xx

11.3.1 405 Method Not Allowed

The method specified in the request is not allowed for the resource identified by the request URI. The response MUST include an Allow header containing a list of valid methods for the requested resource. This status code is also to be used if a request attempts to use a method not indicated during SETUP, e.g., if a RECORD request is issued even though the mode parameter in the Transport header only specified PLAY.

11.3.2 451 Parameter Not Understood

The recipient of the request does not support one or more parameters contained in the request.

11.3.3 452 Conference Not Found

The conference indicated by a Conference header field is unknown to the media server.

11.3.4 453 Not Enough Bandwidth

The request was refused because there was insufficient bandwidth. This may, for example, be the result of a resource reservation failure.

11.3.5 454 Session Not Found

The RTSP session identifier in the Session header is missing, invalid, or has timed out.

11.3.6 455 Method Not Valid in This State

The client or server cannot process this request in its current state. The response SHOULD contain an Allow header to make error recovery easier.

11.3.7 456 Header Field Not Valid for Resource

The server could not act on a required request header. For example, if PLAY contains the Range header field but the stream does not allow seeking.

11.3.8 457 Invalid Range

The Range value given is out of bounds, e.g., beyond the end of the presentation.

11.3.9 458 Parameter Is Read-Only

The parameter to be set by SET_PARAMETER can be read but not modified.

11.3.10 459 Aggregate Operation Not Allowed

The requested method may not be applied on the URL in question since it is an aggregate (presentation) URL. The method may be applied on a stream URL.

11.3.11 460 Only Aggregate Operation Allowed

The requested method may not be applied on the URL in question since it is not an aggregate (presentation) URL. The method may be applied on the presentation URL.

11.3.12 461 Unsupported Transport

The Transport field did not contain a supported transport specification.

11.3.13 462 Destination Unreachable

The data transmission channel could not be established because the client address could not be reached. This error will most likely be the result of a client attempt to place an invalid Destination parameter in the Transport field.

11.3.14 551 Option not supported

An option given in the Require or the Proxy-Require fields was not supported. The Unsupported header should be returned stating the option for which there is no support.

12 Header Field Definitions

HTTP/1.1 [2] or other, non-standard header fields not listed here currently have no well-defined meaning and SHOULD be ignored by the recipient.

Table 3 summarizes the header fields used by RTSP. Type "g" designates general request headers to be found in both requests and responses, type "R" designates request headers, type "r" designates response headers, and type "e" designates entity header fields. Fields marked with "req." in the column labeled "support" MUST be implemented by the recipient for a particular method, while fields marked "opt." are optional. Note that not all fields marked "req." will be sent in every request of this type. The "req." means only that client (for response headers) and server (for request headers) MUST implement the fields. The last column lists the method for which this header field is meaningful; the designation "entity" refers to all methods that return a message body. Within this specification, DESCRIBE and GET_PARAMETER fall into this class.

Header	type	support	methods
Accept	R	opt.	entity
Accept-Encoding	R	opt.	entity
Accept-Language	R	opt.	all
Allow	r	opt.	all
Authorization	R	opt.	all
Bandwidth	R	opt.	all
Blocksize	R	opt.	all but OPTIONS, TEARDOWN
Cache-Control	g	opt.	SETUP
Conference	R	opt.	SETUP
Connection	g	req.	all
Content-Base	e	opt.	entity
Content-Encoding	e	req.	SET_PARAMETER
Content-Encoding	e	req.	DESCRIBE, ANNOUNCE
Content-Language	e	req.	DESCRIBE, ANNOUNCE
Content-Length	e	req.	SET_PARAMETER, ANNOUNCE
Content-Length	e	req.	entity
Content-Location	e	opt.	entity
Content-Type	e	req.	SET_PARAMETER, ANNOUNCE
Content-Type	r	req.	entity
CSeq	g	req.	all
Date	g	opt.	all
Expires	e	opt.	DESCRIBE, ANNOUNCE
From	R	opt.	all
If-Modified-Since	R	opt.	DESCRIBE, SETUP
Last-Modified	e	opt.	entity
Proxy-Authenticate			
Proxy-Require	R	req.	all
Public	r	opt.	all
Range	R	opt.	PLAY, PAUSE, RECORD
Range	r	opt.	PLAY, PAUSE, RECORD
Referer	R	opt.	all
Require	R	req.	all
Retry-After	r	opt.	all
RTP-Info	r	req.	PLAY
Scale	Rr	opt.	PLAY, RECORD
Session	Rr	req.	all but SETUP, OPTIONS
Server	r	opt.	all
Speed	Rr	opt.	PLAY
Transport	Rr	req.	SETUP
Unsupported	r	req.	all
User-Agent	R	opt.	all
Via	g	opt.	all
WWW-Authenticate	r	opt.	all

Overview of RTSP header fields

12.1 Accept

The Accept request-header field can be used to specify certain presentation description content types which are acceptable for the response.

The "level" parameter for presentation descriptions is properly defined as part of the MIME type registration, not here.

See [H14.1] for syntax.

Example of use:
Accept: application/rtsl, application/sdp;level=2

12.2 Accept-Encoding

See [H14.3]

12.3 Accept-Language

See [H14.4]. Note that the language specified applies to the presentation description and any reason phrases, not the media content.

12.4 Allow

The Allow response header field lists the methods supported by the resource identified by the request-URI. The purpose of this field is to strictly inform the recipient of valid methods associated with the resource. An Allow header field must be present in a 405 (Method not allowed) response.

Example of use:
Allow: SETUP, PLAY, RECORD, SET_PARAMETER

12.5 Authorization

See [H14.8]

12.6 Bandwidth

The Bandwidth request header field describes the estimated bandwidth available to the client, expressed as a positive integer and measured in bits per second. The bandwidth available to the client may change during an RTSP session, e.g., due to modem retraining.

```
Bandwidth = "Bandwidth" ":" 1*DIGIT
```

Example:
```
  Bandwidth: 4000
```

12.7 Blocksize

This request header field is sent from the client to the media server asking the server for a particular media packet size. This packet size does not include lower-layer headers such as IP, UDP, or RTP. The server is free to use a blocksize which is lower than the one requested. The server MAY truncate this packet size to the closest multiple of the minimum, media-specific block size, or override it with the media-specific size if necessary. The block size MUST be a positive decimal number, measured in octets. The server only returns an error (416) if the value is syntactically invalid.

12.8 Cache-Control

The Cache-Control general header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.

Cache directives must be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a cache-directive for a specific cache.

Cache-Control should only be specified in a SETUP request and its response. Note: Cache-Control does not govern the caching of responses as for HTTP, but rather of the stream identified by the SETUP request. Responses to RTSP requests are not cacheable, except for responses to DESCRIBE.

```
Cache-Control             =   "Cache-Control" ":" 1#cache-directive
cache-directive           =   cache-request-directive
                          |   cache-response-directive
cache-request-directive   =   "no-cache"
                          |   "max-stale"
                          |   "min-fresh"
                          |   "only-if-cached"
                          |   cache-extension
cache-response-directive  =   "public"
                          |   "private"
                          |   "no-cache"
                          |   "no-transform"
                          |   "must-revalidate"
```

```
                        |   "proxy-revalidate"
                        |   "max-age" "=" delta-seconds
                        |   cache-extension
cache-extension         =   token [ "=" ( token | quoted-string ) ]
```

no-cache:
: Indicates that the media stream MUST NOT be cached anywhere. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

public:
: Indicates that the media stream is cacheable by any cache.

private:
: Indicates that the media stream is intended for a single user and MUST NOT be cached by a shared cache. A private (non-shared) cache may cache the media stream.

no-transform:
: An intermediate cache (proxy) may find it useful to convert the media type of a certain stream. A proxy might, for example, convert between video formats to save cache space or to reduce the amount of traffic on a slow link. Serious operational problems may occur, however, when these transformations have been applied to streams intended for certain kinds of applications. For example, applications for medical imaging, scientific data analysis and those using end-to-end authentication all depend on receiving a stream that is bit-for-bit identical to the original entity-body. Therefore, if a response includes the no-transform directive, an intermediate cache or proxy MUST NOT change the encoding of the stream. Unlike HTTP, RTSP does not provide for partial transformation at this point, e.g., allowing translation into a different language.

only-if-cached:
: In some cases, such as times of extremely poor network connectivity, a client may want a cache to return only those media streams that it currently has stored, and not to receive these from the origin server. To do this, the client may include the only-if-cached directive in a request. If it receives this directive, a cache SHOULD either respond using a cached media stream that is consistent with the other constraints of the request, or respond with a 504 (Gateway Timeout) status. However, if a group of caches is being operated as a unified system with good internal connectivity, such a request MAY be forwarded within that group of caches.

max-stale:
Indicates that the client is willing to accept a media stream that has exceeded its expiration time. If max-stale is assigned a value, then the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If no value is assigned to max-stale, then the client is willing to accept a stale response of any age.

min-fresh:
Indicates that the client is willing to accept a media stream whose freshness lifetime is no less than its current age plus the specified time in seconds. That is, the client wants a response that will still be fresh for at least the specified number of seconds.

must-revalidate:
When the must-revalidate directive is present in a SETUP response received by a cache, that cache MUST NOT use the entry after it becomes stale to respond to a subsequent request without first revalidating it with the origin server. That is, the cache must do an end-to-end revalidation every time, if, based solely on the origin server's Expires, the cached response is stale.)

12.9 Conference

This request header field establishes a logical connection between a pre-established conference and an RTSP stream. The conference-id must not be changed for the same RTSP session.

Conference = "Conference" ":" conference-id Example:
Conference: 199702170042.SAA08642@obiwan.arl.wustl.edu%20Starr

A response code of 452 (452 Conference Not Found) is returned if the conference-id is not valid.

12.10 Connection

See [H14.10]

12.11 Content-Base

See [H14.11]

12.12 Content-Encoding

See [H14.12]

12.13 Content-Language

See [H14.13]

12.14 Content-Length

This field contains the length of the content of the method (i.e. after the double CRLF following the last header). Unlike HTTP, it MUST be included in all messages that carry content beyond the header portion of the message. If it is missing, a default value of zero is assumed. It is interpreted according to [H14.14].

12.15 Content-Location

See [H14.15]

12.16 Content-Type

See [H14.18]. Note that the content types suitable for RTSP are likely to be restricted in practice to presentation descriptions and parameter-value types.

12.17 CSeq

The CSeq field specifies the sequence number for an RTSP request-response pair. This field MUST be present in all requests and responses. For every RTSP request containing the given sequence number, there will be a corresponding response having the same number. Any retransmitted request must contain the same sequence number as the original (i.e. the sequence number is not incremented for retransmissions of the same request).

12.18 Date

See [H14.19].

12.19 Expires

The Expires entity-header field gives a date and time after which the description or media-stream should be considered stale. The interpretation depends on the method:

DESCRIBE response:
The Expires header indicates a date and time after which the description should be considered stale.

A stale cache entry may not normally be returned by a cache (either a proxy cache or an user agent cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the entity). See section 13 for further discussion of the expiration model.

The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time.

The format is an absolute date and time as defined by HTTP-date in [H3.3]; it MUST be in RFC1123-date format:

Expires = "Expires" ":" HTTP-date

An example of its use is

Expires: Thu, 01 Dec 1994 16:00:00 GMT

RTSP/1.0 clients and caches MUST treat other invalid date formats, especially including the value "0", as having occurred in the past (i.e., "already expired").

To mark a response as "already expired," an origin server should use an Expires date that is equal to the Date header value. To mark a response as "never expires," an origin server should use an Expires date approximately one year from the time the response is sent. RTSP/1.0 servers should not send Expires dates more than one year in the future.

The presence of an Expires header field with a date value of some time in the future on a media stream that otherwise would by default be non-cacheable indicates that the media stream is cacheable, unless indicated otherwise by a Cache-Control header field (Section 12.8).

12.20 From

See [H14.22].

12.21 Host

This HTTP request header field is not needed for RTSP. It should be silently ignored if sent.

12.22 If-Match

See [H14.25].

This field is especially useful for ensuring the integrity of the presentation description, in both the case where it is fetched via means external to RTSP (such as HTTP), or in the case where the server implementation is guaranteeing the integrity of the description between the time of the DESCRIBE message and the SETUP message.

The identifier is an opaque identifier, and thus is not specific to any particular session description language.

12.23 If-Modified-Since

The If-Modified-Since request-header field is used with the DESCRIBE and SETUP methods to make them conditional. If the requested variant has not been modified since the time specified in this field, a description will not be returned from the server (DESCRIBE) or a stream will not be set up (SETUP). Instead, a 304 (not modified) response will be returned without any message-body.

```
If-Modified-Since = "If-Modified-Since" ":" HTTP-date
```

An example of the field is:

```
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

12.24 Last-Modified

The Last-Modified entity-header field indicates the date and time at which the origin server believes the presentation description or media stream was last modified. See [H14.29]. For the methods DESCRIBE or ANNOUNCE, the header field indicates the last modification date and time of the description, for SETUP that of the media stream.

12.25 Location

See [H14.30].

12.26 Proxy-Authenticate

See [H14.33].

12.27 Proxy-Require

The Proxy-Require header is used to indicate proxy-sensitive features that MUST be supported by the proxy. Any Proxy-Require header features that are not supported by the proxy MUST be negatively acknowledged by the proxy to the client if not supported. Servers

should treat this field identically to the Require field.

See Section 12.32 for more details on the mechanics of this message and a usage example.

12.28 Public

See [H14.35].

12.29 Range

This request and response header field specifies a range of time. The range can be specified in a number of units. This specification defines the smpte (Section 3.5), npt (Section 3.6), and clock (Section 3.7) range units. Within RTSP, byte ranges [H14.36.1] are not meaningful and MUST NOT be used. The header may also contain a time parameter in UTC, specifying the time at which the operation is to be made effective. Servers supporting the Range header MUST understand the NPT range format and SHOULD understand the SMPTE range format. The Range response header indicates what range of time is actually being played or recorded. If the Range header is given in a time format that is not understood, the recipient should return "501 Not Implemented".

Ranges are half-open intervals, including the lower point, but excluding the upper point. In other words, a range of a-b starts exactly at time a, but stops just before b. Only the start time of a media unit such as a video or audio frame is relevant. As an example, assume that video frames are generated every 40 ms. A range of 10.0-10.1 would include a video frame starting at 10.0 or later time and would include a video frame starting at 10.08, even though it lasted beyond the interval. A range of 10.0-10.08, on the other hand, would exclude the frame at 10.08.

```
Range              = "Range" ":" 1\#ranges-specifier
                         [ ";" "time" "=" utc-time ]
ranges-specifier = npt-range | utc-range | smpte-range
```

Example:

```
Range: clock=19960213T143205Z-;time=19970123T143720Z
```

The notation is similar to that used for the HTTP/1.1 [2] byte-range header. It allows clients to select an excerpt from the media object, and to play from a given point to the end as well as from the current location to a given point. The start of playback can be scheduled for any time in the future, although a server may refuse to keep server resources for extended idle periods.

12.30 Referer

See [H14.37]. The URL refers to that of the presentation description, typically retrieved via HTTP.

12.31 Retry-After

See [H14.38].

12.32 Require

The Require header is used by clients to query the server about options that it may or may not support. The server MUST respond to this header by using the Unsupported header to negatively acknowledge those options which are NOT supported.

This is to make sure that the client-server interaction will proceed without delay when all options are understood by both sides, and only slow down if options are not understood (as in the case above). For a well-matched client-server pair, the interaction proceeds quickly, saving a round-trip often required by negotiation mechanisms. In addition, it also removes state ambiguity when the client requires features that the server does not understand.

```
Require =   "Require" ":"  1#option-tag
```

Example:

```
  C->S:   SETUP rtsp://server.com/foo/bar/baz.rm RTSP/1.0
          CSeq: 302
          Require: funky-feature
          Funky-Parameter: funkystuff

  S->C:   RTSP/1.0 551 Option not supported
          CSeq: 302
          Unsupported: funky-feature

  C->S:   SETUP rtsp://server.com/foo/bar/baz.rm RTSP/1.0
          CSeq: 303

  S->C:   RTSP/1.0 200 OK
          CSeq: 303
```

In this example, "funky-feature" is the feature tag which indicates to the client that the fictional Funky-Parameter field is required. The relationship between "funky-feature" and Funky-Parameter is not communicated via the RTSP exchange, since that relationship is an immutable property of "funky-feature" and thus should not be transmitted with every exchange.

Proxies and other intermediary devices SHOULD ignore features that are not understood in this field. If a particular extension requires that intermediate devices support it, the extension should be tagged in the Proxy-Require field instead (see Section 12.27).

12.33 RTP-Info

This field is used to set RTP-specific parameters in the PLAY response.

url:
: Indicates the stream URL which for which the following RTP parameters correspond.

seq:
: Indicates the sequence number of the first packet of the stream. This allows clients to gracefully deal with packets when seeking. The client uses this value to differentiate packets that originated before the seek from packets that originated after the seek.

rtptime:
: Indicates the RTP timestamp corresponding to the time value in the Range response header. (Note: For aggregate control, a particular stream may not actually generate a packet for the Range time value returned or implied. Thus, there is no guarantee that the packet with the sequence number indicated by seq actually has the timestamp indicated by rtptime.) The client uses this value to calculate the mapping of RTP time to NPT.

A mapping from RTP timestamps to NTP timestamps (wall clock) is available via RTCP. However, this information is not sufficient to generate a mapping from RTP timestamps to NPT. Furthermore, in order to ensure that this information is available at the necessary time (immediately at startup or after a seek), and that it is delivered reliably, this mapping is placed in the RTSP control channel.

In order to compensate for drift for long, uninterrupted presentations, RTSP clients should additionally map NPT to NTP, using initial RTCP sender reports to do the mapping, and later reports to check drift against the mapping.

Syntax:

```
RTP-Info        = "RTP-Info" ":" 1#stream-url 1*parameter
stream-url      = "url" "=" url
parameter       = ";" "seq" "=" 1*DIGIT
                | ";" "rtptime" "=" 1*DIGIT
```

Example:

```
RTP-Info: url=rtsp://foo.com/bar.avi/streamid=0;seq=45102,
          url=rtsp://foo.com/bar.avi/streamid=1;seq=30211
```

12.34 Scale

A scale value of 1 indicates normal play or record at the normal forward viewing rate. If not 1, the value corresponds to the rate with respect to normal viewing rate. For example, a ratio of 2 indicates twice the normal viewing rate ("fast forward") and a ratio of 0.5 indicates half the normal viewing rate. In other words, a ratio of 2 has normal play time increase at twice the wallclock rate. For every second of elapsed (wallclock) time, 2 seconds of content will be delivered. A negative value indicates reverse direction.

Unless requested otherwise by the Speed parameter, the data rate SHOULD not be changed. Implementation of scale changes depends on the server and media type. For video, a server may, for example, deliver only key frames or selected key frames. For audio, it may time-scale the audio while preserving pitch or, less desirably, deliver fragments of audio.

The server should try to approximate the viewing rate, but may restrict the range of scale values that it supports. The response MUST contain the actual scale value chosen by the server.

If the request contains a Range parameter, the new scale value will take effect at that time.

```
Scale = "Scale" ":" [ "-" ] 1*DIGIT [ "." *DIGIT ]
```

Example of playing in reverse at 3.5 times normal rate:

```
Scale: -3.5
```

12.35 Speed

This request header fields parameter requests the server to deliver data to the client at a particular speed, contingent on the server's ability and desire to serve the media stream at the given speed. Implementation by the server is OPTIONAL. The default is the bit rate of the stream.

The parameter value is expressed as a decimal ratio, e.g., a value of 2.0 indicates that data is to be delivered twice as fast as normal. A speed of zero is invalid. If the request contains a Range parameter, the new speed value will take effect at that time.

```
Speed = "Speed" ":" 1*DIGIT [ "." *DIGIT ]
```

Example:

```
  Speed: 2.5
```

Use of this field changes the bandwidth used for data delivery. It is meant for use in specific circumstances where preview of the presentation at a higher or lower rate is necessary. Implementors should keep in mind that bandwidth for the session may be negotiated beforehand (by means other than RTSP), and therefore re-negotiation may be necessary. When data is delivered over UDP, it is highly recommended that means such as RTCP be used to track packet loss rates.

12.36 Server

See [H14.39]

12.37 Session

This request and response header field identifies an RTSP session started by the media server in a SETUP response and concluded by TEARDOWN on the presentation URL. The session identifier is chosen by the media server (see Section 3.4). Once a client receives a Session identifier, it MUST return it for any request related to that session. A server does not have to set up a session identifier if it has other means of identifying a session, such as dynamically generated URLs.

```
Session  = "Session" ":" session-id [ ";" "timeout" "=" delta-seconds ]
```

The timeout parameter is only allowed in a response header. The server uses it to indicate to the client how long the server is prepared to wait between RTSP commands before closing the session due to lack of activity (see Section A). The timeout is measured in

seconds, with a default of 60 seconds (1 minute).

Note that a session identifier identifies a RTSP session across transport sessions or connections. Control messages for more than one RTSP URL may be sent within a single RTSP session. Hence, it is possible that clients use the same session for controlling many streams constituting a presentation, as long as all the streams come from the same server. (See example in Section 14). However, multiple "user" sessions for the same URL from the same client MUST use different session identifiers.

The session identifier is needed to distinguish several delivery requests for the same URL coming from the same client.

The response 454 (Session Not Found) is returned if the session identifier is invalid.

12.38 Timestamp

The timestamp general header describes when the client sent the request to the server. The value of the timestamp is of significance only to the client and may use any timescale. The server MUST echo the exact same value and MAY, if it has accurate information about this, add a floating point number indicating the number of seconds that has elapsed since it has received the request. The timestamp is used by the client to compute the round-trip time to the server so that it can adjust the timeout value for retransmissions.

```
Timestamp  = "Timestamp" ":" *(DIGIT) [ "." *(DIGIT) ] [ delay ]
delay      =  *(DIGIT) [ "." *(DIGIT) ]
```

12.39 Transport

This request header indicates which transport protocol is to be used and configures its parameters such as destination address, compression, multicast time-to-live and destination port for a single stream. It sets those values not already determined by a presentation description.

Transports are comma separated, listed in order of preference. Parameters may be added to each transport, separated by a semicolon.

The Transport header MAY also be used to change certain transport parameters. A server MAY refuse to change parameters of an existing stream.

The server MAY return a Transport response header in the response to indicate the values actually chosen.

A Transport request header field may contain a list of transport options acceptable to the client. In that case, the server MUST return a single option which was actually chosen.

The syntax for the transport specifier is

transport/profile/lower-transport.

The default value for the "lower-transport" parameters is specific to the profile. For RTP/AVP, the default is UDP.

Below are the configuration parameters associated with transport:

General parameters:

unicast | multicast:
: mutually exclusive indication of whether unicast or multicast delivery will be attempted. Default value is multicast. Clients that are capable of handling both unicast and multicast transmission MUST indicate such capability by including two full transport-specs with separate parameters for each.

destination:
: The address to which a stream will be sent. The client may specify the multicast address with the destination parameter. To avoid becoming the unwitting perpetrator of a remote-controlled denial-of-service attack, a server SHOULD authenticate the client and SHOULD log such attempts before allowing the client to direct a media stream to an address not chosen by the server. This is particularly important if RTSP commands are issued via UDP, but implementations cannot rely on TCP as reliable means of client identification by itself. A server SHOULD not allow a client to direct media streams to an address that differs from the address commands are coming from.

source:
: If the source address for the stream is different than can be derived from the RTSP endpoint address (the server in playback or the client in recording), the source MAY be specified.

This information may also be available through SDP. However, since this is more a feature of transport than media initialization, the authoritative source for this information should be in the SETUP response.

layers:
The number of multicast layers to be used for this media stream. The layers are sent to consecutive addresses starting at the destination address.

mode:
The mode parameter indicates the methods to be supported for this session. Valid values are PLAY and RECORD. If not provided, the default is PLAY.

append:
If the mode parameter includes RECORD, the append parameter indicates that the media data should append to the existing resource rather than overwrite it. If appending is requested and the server does not support this, it MUST refuse the request rather than overwrite the resource identified by the URI. The append parameter is ignored if the mode parameter does not contain RECORD.

interleaved:
The interleaved parameter implies mixing the media stream with the control stream in whatever protocol is being used by the control stream, using the mechanism defined in Section 10.12. The argument provides the channel number to be used in the $ statement. This parameter may be specified as a range, e.g., interleaved=4-5 in cases where the transport choice for the media stream requires it.

This allows RTP/RTCP to be handled similarly to the way that it is done with UDP, i.e., one channel for RTP and the other for RTCP.

Multicast specific:

ttl:
multicast time-to-live

RTP Specific:

port:
This parameter provides the RTP/RTCP port pair for a multicast session. It is specified as a range, e.g., port=3456-3457.

client_port:
This parameter provides the unicast RTP/RTCP port pair on which the client has chosen to receive media data and control information. It is specified as a range, e.g., client_port=3456-3457.

server_port:
This parameter provides the unicast RTP/RTCP port pair on which the server has chosen to receive media data and control information. It is specified as a range, e.g., server_port=3456-3457.

ssrc:
The ssrc parameter indicates the RTP SSRC [24, Sec. 3] value that should be (request) or will be (response) used by the media server. This parameter is only valid for unicast transmission. It identifies the synchronization source to be associated with the media stream.

```
Transport           =    "Transport" ":"
                         1\#transport-spec
transport-spec      =    transport-protocol/profile[/lower-transport]
                         *parameter
transport-protocol  =    "RTP"
profile             =    "AVP"
lower-transport     =    "TCP" | "UDP"
parameter           =    ( "unicast" | "multicast" )
                    |    ";" "destination" [ "=" address ]
                    |    ";" "interleaved" "=" channel [ "-" channel ]
                    |    ";" "append"
                    |    ";" "ttl" "=" ttl
                    |    ";" "layers" "=" 1*DIGIT
                    |    ";" "port" "=" port [ "-" port ]
                    |    ";" "client_port" "=" port [ "-" port ]
                    |    ";" "server_port" "=" port [ "-" port ]
                    |    ";" "ssrc" "=" ssrc
                    |    ";" "mode" = <"> 1\#mode <">
ttl                 =    1*3(DIGIT)
port                =    1*5(DIGIT)
ssrc                =    8*8(HEX)
channel             =    1*3(DIGIT)
address             =    host
mode                =    <"> *Method <"> | Method
```

Example:

```
  Transport: RTP/AVP;multicast;ttl=127;mode="PLAY",
             RTP/AVP;unicast;client_port=3456-3457;mode="PLAY"
```

The Transport header is restricted to describing a single RTP stream. (RTSP can also control multiple streams as a single entity.) Making it part of RTSP rather than relying on a multitude of session description formats greatly simplifies designs of firewalls.

12.40 Unsupported

The Unsupported response header lists the features not supported by the server. In the case where the feature was specified via the Proxy-Require field (Section 12.32), if there is a proxy on the path between the client and the server, the proxy MUST insert a message reply with an error message "551 Option Not Supported".

See Section 12.32 for a usage example.

12.41 User-Agent

See [H14.42]

12.42 Vary

See [H14.43]

12.43 Via

See [H14.44].

12.44 WWW-Authentica

See [H14.46].

13 Caching

In HTTP, response-request pairs are cached. RTSP differs significantly in that respect. Responses are not cacheable, with the exception of the presentation description returned by DESCRIBE or included with ANNOUNCE. (Since the responses for anything but DESCRIBE and GET_PARAMETER do not return any data, caching is not really an issue for these requests.) However, it is desirable for the continuous media data, typically delivered out-of-band with respect to RTSP, to be cached, as well as the session description.

On receiving a SETUP or PLAY request, a proxy ascertains whether it has an up-to-date copy of the continuous media content and its description. It can determine whether the copy is up-to-date by issuing a SETUP or DESCRIBE request, respectively, and comparing the Last-Modified header with that of the cached copy. If the copy is not up-to-date, it modifies the SETUP transport parameters as appropriate and forwards the request to the origin server. Subsequent control commands such as PLAY or PAUSE then pass the proxy unmodified. The proxy delivers the continuous media data to the client, while possibly making a local copy for later reuse. The exact behavior allowed to the cache is given by the cache-response directives

described in Section 12.8. A cache MUST answer any DESCRIBE requests if it is currently serving the stream to the requestor, as it is possible that low-level details of the stream description may have changed on the origin-server.

Note that an RTSP cache, unlike the HTTP cache, is of the "cut-through" variety. Rather than retrieving the whole resource from the origin server, the cache simply copies the streaming data as it passes by on its way to the client. Thus, it does not introduce additional latency.

To the client, an RTSP proxy cache appears like a regular media server, to the media origin server like a client. Just as an HTTP cache has to store the content type, content language, and so on for the objects it caches, a media cache has to store the presentation description. Typically, a cache eliminates all transport-references (that is, multicast information) from the presentation description, since these are independent of the data delivery from the cache to the client. Information on the encodings remains the same. If the cache is able to translate the cached media data, it would create a new presentation description with all the encoding possibilities it can offer.

14 Examples

The following examples refer to stream description formats that are not standards, such as RTSL. The following examples are not to be used as a reference for those formats.

14.1 Media on Demand (Unicast)

Client C requests a movie from media servers A (audio.example.com) and V (video.example.com). The media description is stored on a web server W . The media description contains descriptions of the presentation and all its streams, including the codecs that are available, dynamic RTP payload types, the protocol stack, and content information such as language or copyright restrictions. It may also give an indication about the timeline of the movie.

In this example, the client is only interested in the last part of the movie.

```
C->W: GET /twister.sdp HTTP/1.1
      Host: www.example.com
      Accept: application/sdp

W->C: HTTP/1.0 200 OK
      Content-Type: application/sdp
```

```
        v=0
        o=- 2890844526 2890842807 IN IP4 192.16.24.202
        s=RTSP Session
        m=audio 0 RTP/AVP 0
        a=control:rtsp://audio.example.com/twister/audio.en
        m=video 0 RTP/AVP 31
        a=control:rtsp://video.example.com/twister/video

   C->A: SETUP rtsp://audio.example.com/twister/audio.en RTSP/1.0
        CSeq: 1
        Transport: RTP/AVP/UDP;unicast;client_port=3056-3057

   A->C: RTSP/1.0 200 OK
        CSeq: 1
        Session: 12345678
        Transport: RTP/AVP/UDP;unicast;client_port=3056-3057;
                   server_port=5000-5001

   C->V: SETUP rtsp://video.example.com/twister/video RTSP/1.0
        CSeq: 1
        Transport: RTP/AVP/UDP;unicast;client_port=3058-3059

   V->C: RTSP/1.0 200 OK
        CSeq: 1
        Session: 23456789
        Transport: RTP/AVP/UDP;unicast;client_port=3058-3059;
                   server_port=5002-5003

   C->V: PLAY rtsp://video.example.com/twister/video RTSP/1.0
        CSeq: 2
        Session: 23456789
        Range: smpte=0:10:00-

   V->C: RTSP/1.0 200 OK
        CSeq: 2
        Session: 23456789
        Range: smpte=0:10:00-0:20:00
        RTP-Info: url=rtsp://video.example.com/twister/video;
          seq=12312232;rtptime=78712811

   C->A: PLAY rtsp://audio.example.com/twister/audio.en RTSP/1.0
        CSeq: 2
        Session: 12345678
        Range: smpte=0:10:00-

   A->C: RTSP/1.0 200 OK
        CSeq: 2
        Session: 12345678
```

```
        Range: smpte=0:10:00-0:20:00
        RTP-Info: url=rtsp://audio.example.com/twister/audio.en;
          seq=876655;rtptime=1032181

C->A:   TEARDOWN rtsp://audio.example.com/twister/audio.en RTSP/1.0
        CSeq: 3
        Session: 12345678

A->C:   RTSP/1.0 200 OK
        CSeq: 3

C->V:   TEARDOWN rtsp://video.example.com/twister/video RTSP/1.0
        CSeq: 3
        Session: 23456789

V->C:   RTSP/1.0 200 OK
        CSeq: 3
```

Even though the audio and video track are on two different servers, and may start at slightly different times and may drift with respect to each other, the client can synchronize the two using standard RTP methods, in particular the time scale contained in the RTCP sender reports.

14.2 Streaming of a Container file

For purposes of this example, a container file is a storage entity in which multiple continuous media types pertaining to the same end-user presentation are present. In effect, the container file represents an RTSP presentation, with each of its components being RTSP streams. Container files are a widely used means to store such presentations. While the components are transported as independent streams, it is desirable to maintain a common context for those streams at the server end.

> This enables the server to keep a single storage handle open easily. It also allows treating all the streams equally in case of any prioritization of streams by the server.

It is also possible that the presentation author may wish to prevent selective retrieval of the streams by the client in order to preserve the artistic effect of the combined media presentation. Similarly, in such a tightly bound presentation, it is desirable to be able to control all the streams via a single control message using an aggregate URL.

The following is an example of using a single RTSP session to control multiple streams. It also illustrates the use of aggregate URLs.

Client C requests a presentation from media server M . The movie is stored in a container file. The client has obtained an RTSP URL to the container file.

```
C->M: DESCRIBE rtsp://foo/twister RTSP/1.0
      CSeq: 1

M->C: RTSP/1.0 200 OK
      CSeq: 1
      Content-Type: application/sdp
      Content-Length: 164

      v=0
      o=- 2890844256 2890842807 IN IP4 172.16.2.93
      s=RTSP Session
      i=An Example of RTSP Session Usage
      a=control:rtsp://foo/twister
      t=0 0
      m=audio 0 RTP/AVP 0
      a=control:rtsp://foo/twister/audio
      m=video 0 RTP/AVP 26
      a=control:rtsp://foo/twister/video

C->M: SETUP rtsp://foo/twister/audio RTSP/1.0
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8000-8001

M->C: RTSP/1.0 200 OK
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8000-8001;
                 server_port=9000-9001
      Session: 12345678

C->M: SETUP rtsp://foo/twister/video RTSP/1.0
      CSeq: 3
      Transport: RTP/AVP;unicast;client_port=8002-8003
      Session: 12345678

M->C: RTSP/1.0 200 OK
      CSeq: 3
      Transport: RTP/AVP;unicast;client_port=8002-8003;
                 server_port=9004-9005
      Session: 12345678

C->M: PLAY rtsp://foo/twister RTSP/1.0
      CSeq: 4
      Range: npt=0-
      Session: 12345678
```

```
M->C: RTSP/1.0 200 OK
      CSeq: 4
      Session: 12345678
      RTP-Info: url=rtsp://foo/twister/video;
        seq=9810092;rtptime=3450012

C->M: PAUSE rtsp://foo/twister/video RTSP/1.0
      CSeq: 5
      Session: 12345678

M->C: RTSP/1.0 460 Only aggregate operation allowed
      CSeq: 5

C->M: PAUSE rtsp://foo/twister RTSP/1.0
      CSeq: 6
      Session: 12345678

M->C: RTSP/1.0 200 OK
      CSeq: 6
      Session: 12345678

C->M: SETUP rtsp://foo/twister RTSP/1.0
      CSeq: 7
      Transport: RTP/AVP;unicast;client_port=10000

M->C: RTSP/1.0 459 Aggregate operation not allowed
      CSeq: 7
```

In the first instance of failure, the client tries to pause one stream (in this case video) of the presentation. This is disallowed for that presentation by the server. In the second instance, the aggregate URL may not be used for SETUP and one control message is required per stream to set up transport parameters.

This keeps the syntax of the Transport header simple and allows easy parsing of transport information by firewalls.

14.3 Single Stream Container Files

Some RTSP servers may treat all files as though they are "container files", yet other servers may not support such a concept. Because of this, clients SHOULD use the rules set forth in the session description for request URLs, rather than assuming that a consistent URL may always be used throughout. Here's an example of how a multi-stream server might expect a single-stream file to be served:

```
Accept: application/x-rtsp-mh, application/sdp
```

```
      CSeq: 1

S->C  RTSP/1.0 200 OK
      CSeq: 1
      Content-base: rtsp://foo.com/test.wav/
      Content-type: application/sdp
      Content-length: 48

      v=0
      o=- 872653257 872653257 IN IP4 172.16.2.187
      s=mu-law wave file
      i=audio test
      t=0 0
      m=audio 0 RTP/AVP 0
      a=control:streamid=0

C->S  SETUP rtsp://foo.com/test.wav/streamid=0 RTSP/1.0
      Transport: RTP/AVP/UDP;unicast;
                 client_port=6970-6971;mode=play
      CSeq: 2

S->C  RTSP/1.0 200 OK
      Transport: RTP/AVP/UDP;unicast;client_port=6970-6971;
                 server_port=6970-6971;mode=play
      CSeq: 2
      Session: 2034820394

C->S  PLAY rtsp://foo.com/test.wav RTSP/1.0
      CSeq: 3
      Session: 2034820394

S->C  RTSP/1.0 200 OK
      CSeq: 3
      Session: 2034820394
      RTP-Info: url=rtsp://foo.com/test.wav/streamid=0;
        seq=981888;rtptime=3781123
```

Note the different URL in the SETUP command, and then the switch back to the aggregate URL in the PLAY command. This makes complete sense when there are multiple streams with aggregate control, but is less than intuitive in the special case where the number of streams is one.

In this special case, it is recommended that servers be forgiving of implementations that send:

```
C->S  PLAY rtsp://foo.com/test.wav/streamid=0 RTSP/1.0
      CSeq: 3
```

In the worst case, servers should send back:

```
S->C  RTSP/1.0 460 Only aggregate operation allowed
      CSeq: 3
```

One would also hope that server implementations are also forgiving of the following:

```
C->S  SETUP rtsp://foo.com/test.wav RTSP/1.0
      Transport: rtp/avp/udp;client_port=6970-6971;mode=play
      CSeq: 2
```

Since there is only a single stream in this file, it's not ambiguous what this means.

14.4 Live Media Presentation Using Multicast

The media server M chooses the multicast address and port. Here, we assume that the web server only contains a pointer to the full description, while the media server M maintains the full description.

```
C->W: GET /concert.sdp HTTP/1.1
      Host: www.example.com

W->C: HTTP/1.1 200 OK
      Content-Type: application/x-rtsl

      <session>
        <track src="rtsp://live.example.com/concert/audio">
      </session>

C->M: DESCRIBE rtsp://live.example.com/concert/audio RTSP/1.0
      CSeq: 1

M->C: RTSP/1.0 200 OK
      CSeq: 1
      Content-Type: application/sdp
      Content-Length: 44

      v=0
      o=- 2890844526 2890842807 IN IP4 192.16.24.202
      s=RTSP Session
      m=audio 3456 RTP/AVP 0
      a=control:rtsp://live.example.com/concert/audio
      c=IN IP4 224.2.0.1/16

C->M: SETUP rtsp://live.example.com/concert/audio RTSP/1.0
      CSeq: 2
```

```
          Transport: RTP/AVP;multicast

     M->C: RTSP/1.0 200 OK
          CSeq: 2
          Transport: RTP/AVP;multicast;destination=224.2.0.1;
                     port=3456-3457;ttl=16
          Session: 0456804596

     C->M: PLAY rtsp://live.example.com/concert/audio RTSP/1.0
          CSeq: 3
          Session: 0456804596

     M->C: RTSP/1.0 200 OK
          CSeq: 3
          Session: 0456804596
```

14.5 Playing media into an existing session

A conference participant C wants to have the media server M play back a demo tape into an existing conference. C indicates to the media server that the network addresses and encryption keys are already given by the conference, so they should not be chosen by the server. The example omits the simple ACK responses.

```
     C->M: DESCRIBE rtsp://server.example.com/demo/548/sound RTSP/1.0
          CSeq: 1
          Accept: application/sdp

     M->C: RTSP/1.0 200 1 OK
          Content-type: application/sdp
          Content-Length: 44

          v=0
          o=- 2890844526 2890842807 IN IP4 192.16.24.202
          s=RTSP Session
          i=See above
          t=0 0
          m=audio 0 RTP/AVP 0

     C->M: SETUP rtsp://server.example.com/demo/548/sound RTSP/1.0
          CSeq: 2
          Transport: RTP/AVP;multicast;destination=225.219.201.15;
                     port=7000-7001;ttl=127
          Conference: 199702170042.SAA08642@obiwan.arl.wustl.edu%20Starr

     M->C: RTSP/1.0 200 OK
          CSeq: 2
          Transport: RTP/AVP;multicast;destination=225.219.201.15;
```

```
                port=7000-7001;ttl=127
        Session: 91389234234
        Conference: 199702170042.SAA08642@obiwan.arl.wustl.edu%20Starr

C->M:   PLAY rtsp://server.example.com/demo/548/sound RTSP/1.0
        CSeq: 3
        Session: 91389234234

M->C:   RTSP/1.0 200 OK
        CSeq: 3
```

14.6 Recording

The conference participant client C asks the media server M to record the audio and video portions of a meeting. The client uses the ANNOUNCE method to provide meta-information about the recorded session to the server.

```
C->M:   ANNOUNCE rtsp://server.example.com/meeting RTSP/1.0
        CSeq: 90
        Content-Type: application/sdp
        Content-Length: 121

        v=0
        o=camera1 3080117314 3080118787 IN IP4 195.27.192.36
        s=IETF Meeting, Munich - 1
        i=The thirty-ninth IETF meeting will be held in Munich, Germany
        u=http://www.ietf.org/meetings/Munich.html
        e=IETF Channel 1 <ietf39-mbone@uni-koeln.de>
        p=IETF Channel 1 +49-172-2312 451
        c=IN IP4 224.0.1.11/127
        t=3080271600 3080703600
        a=tool:sdr v2.4a6
        a=type:test
        m=audio 21010 RTP/AVP 5
        c=IN IP4 224.0.1.11/127
        a=ptime:40
        m=video 61010 RTP/AVP 31
        c=IN IP4 224.0.1.12/127

M->C:   RTSP/1.0 200 OK
        CSeq: 90

C->M:   SETUP rtsp://server.example.com/meeting/audiotrack RTSP/1.0
        CSeq: 91
        Transport: RTP/AVP;multicast;destination=224.0.1.11;
                   port=21010-21011;mode=record;ttl=127
```

```
M->C: RTSP/1.0 200 OK
      CSeq: 91
      Session: 50887676
      Transport: RTP/AVP;multicast;destination=224.0.1.11;
                 port=21010-21011;mode=record;ttl=127

C->M: SETUP rtsp://server.example.com/meeting/videotrack RTSP/1.0
      CSeq: 92
      Session: 50887676
      Transport: RTP/AVP;multicast;destination=224.0.1.12;
                 port=61010-61011;mode=record;ttl=127

M->C: RTSP/1.0 200 OK
      CSeq: 92
      Transport: RTP/AVP;multicast;destination=224.0.1.12;
                 port=61010-61011;mode=record;ttl=127

C->M: RECORD rtsp://server.example.com/meeting RTSP/1.0
      CSeq: 93
      Session: 50887676
      Range: clock=19961110T1925-19961110T2015

M->C: RTSP/1.0 200 OK
      CSeq: 93
```

15 Syntax

The RTSP syntax is described in an augmented Backus-Naur form (BNF) as used in RFC 2068 [2].

15.1 Base Syntax

```
OCTET          =    <any 8-bit sequence of data>
CHAR           =    <any US-ASCII character (octets 0 - 127)>
UPALPHA        =    <any US-ASCII uppercase letter "A".."Z">
LOALPHA        =    <any US-ASCII lowercase letter "a".."z">
ALPHA          =    UPALPHA | LOALPHA

DIGIT          =    <any US-ASCII digit "0".."9">
CTL            =    <any US-ASCII control character
                     (octets 0 - 31) and DEL (127)>
CR             =    <US-ASCII CR, carriage return (13)>
LF             =    <US-ASCII LF, linefeed (10)>

SP             =    <US-ASCII SP, space (32)>
HT             =    <US-ASCII HT, horizontal-tab (9)>
<">            =    <US-ASCII double-quote mark (34)>
CRLF           =    CR LF
```

```
LWS                 =       [CRLF] 1*( SP | HT )
TEXT                =       <any OCTET except CTLs>
tspecials           =       "(" | ")" | "<" | ">" | "@"
                    |        "," | ";" | ":" | "\" | <">
                    |        "/" | "[" | "]" | "?" | "="
                    |        "{" | "}" | SP | HT

token               =       1*<any CHAR except CTLs or tspecials>
quoted-string       =       ( <"> *(qdtext) <"> )
qdtext              =       <any TEXT except <">>
quoted-pair         =       "\" CHAR

message-header      =       field-name ":" [ field-value ] CRLF
field-name          =       token
field-value         =       *( field-content | LWS )
field-content       =       <the OCTETs making up the field-value and
                             consisting of either *TEXT or
                             combinations of token, tspecials, and
                             quoted-string>

safe                =  "\$" | "-" | "_" | "." | "+"
extra               =  "!" | "*" | "$'$" | "(" | ")" | ","

hex                 =  DIGIT | "A" | "B" | "C" | "D" | "E" | "F" |
                       "a" | "b" | "c" | "d" | "e" | "f"
escape              =  "\%" hex hex
reserved            =  ";" | "/" | "?" | ":" | "@" | "&" | "="

unreserved          =  alpha | digit | safe | extra
xchar               =  unreserved | reserved | escape
```

16 Security Considerations

Because of the similarity in syntax and usage between RTSP servers and HTTP servers, the security considerations outlined in [H15] apply. Specifically, please note the following:

Authentication Mechanisms:

RTSP and HTTP share common authentication schemes, and thus should follow the same prescriptions with regards to authentication. See [H15.1] for client authentication issues, and [H15.2] for issues regarding support for multiple authentication mechanisms.

Abuse of Server Log Information:

RTSP and HTTP servers will presumably have similar logging mechanisms, and thus should be equally guarded in protecting the contents of those logs, thus protecting the privacy of the

users of the servers. See [H15.3] for HTTP server recommendations regarding server logs.

Transfer of Sensitive Information:
There is no reason to believe that information transferred via RTSP may be any less sensitive than that normally transmitted via HTTP. Therefore, all of the precautions regarding the protection of data privacy and user privacy apply to implementors of RTSP clients, servers, and proxies. See [H15.4] for further details.

Attacks Based On File and Path Names:
Though RTSP URLs are opaque handles that do not necessarily have file system semantics, it is anticipated that many implementations will translate portions of the request URLs directly to file system calls. In such cases, file systems SHOULD follow the precautions outlined in [H15.5], such as checking for ".." in path components.

Personal Information:
RTSP clients are often privy to the same information that HTTP clients are (user name, location, etc.) and thus should be equally. See [H15.6] for further recommendations.

Privacy Issues Connected to Accept Headers:
Since may of the same "Accept" headers exist in RTSP as in HTTP, the same caveats outlined in [H15.7] with regards to their use should be followed.

DNS Spoofing:
Presumably, given the longer connection times typically associated to RTSP sessions relative to HTTP sessions, RTSP client DNS optimizations should be less prevalent. Nonetheless, the recommendations provided in [H15.8] are still relevant to any implementation which attempts to rely on a DNS-to-IP mapping to hold beyond a single use of the mapping.

Location Headers and Spoofing:
If a single server supports multiple organizations that do not trust one another, then it must check the values of Location and Content-Location headers in responses that are generated under control of said organizations to make sure that they do not attempt to invalidate resources over which they have no authority. ([H15.9])

In addition to the recommendations in the current HTTP specification (RFC 2068 [2], as of this writing), future HTTP specifications may provide additional guidance on security issues.

The following are added considerations for RTSP implementations.

Concentrated denial-of-service attack:
: The protocol offers the opportunity for a remote-controlled denial-of-service attack. The attacker may initiate traffic flows to one or more IP addresses by specifying them as the destination in SETUP requests. While the attacker's IP address may be known in this case, this is not always useful in prevention of more attacks or ascertaining the attackers identity. Thus, an RTSP server SHOULD only allow client-specified destinations for RTSP-initiated traffic flows if the server has verified the client's identity, either against a database of known users using RTSP authentication mechanisms (preferably digest authentication or stronger), or other secure means.

Session hijacking:
: Since there is no relation between a transport layer connection and an RTSP session, it is possible for a malicious client to issue requests with random session identifiers which would affect unsuspecting clients. The server SHOULD use a large, random and non-sequential session identifier to minimize the possibility of this kind of attack.

Authentication:
: Servers SHOULD implement both basic and digest [8] authentication. In environments requiring tighter security for the control messages, the RTSP control stream may be encrypted.

Stream issues:
: RTSP only provides for stream control. Stream delivery issues are not covered in this section, nor in the rest of this memo. RTSP implementations will most likely rely on other protocols such as RTP, IP multicast, RSVP and IGMP, and should address security considerations brought up in those and other applicable specifications.

Persistently suspicious behavior:
: RTSP servers SHOULD return error code 403 (Forbidden) upon receiving a single instance of behavior which is deemed a security risk. RTSP servers SHOULD also be aware of attempts to probe the server for weaknesses and entry points and MAY arbitrarily disconnect and ignore further requests clients which are deemed to be in violation of local security policy.

Appendix A: RTSP Protocol State Machines

The RTSP client and server state machines describe the behavior of the protocol from RTSP session initialization through RTSP session termination.

State is defined on a per object basis. An object is uniquely identified by the stream URL and the RTSP session identifier. Any request/reply using aggregate URLs denoting RTSP presentations composed of multiple streams will have an effect on the individual states of all the streams. For example, if the presentation /movie contains two streams, /movie/audio and /movie/video, then the following command:

```
PLAY rtsp://foo.com/movie RTSP/1.0
CSeq: 559
Session: 12345678
```

will have an effect on the states of movie/audio and movie/video.

This example does not imply a standard way to represent streams in URLs or a relation to the filesystem. See Section 3.2.

The requests OPTIONS, ANNOUNCE, DESCRIBE, GET_PARAMETER, SET_PARAMETER do not have any effect on client or server state and are therefore not listed in the state tables.

A.1 Client State Machine

The client can assume the following states:

Init:
SETUP has been sent, waiting for reply.

Ready:
SETUP reply received or PAUSE reply received while in Playing state.

Playing:
PLAY reply received

Recording:
RECORD reply received

In general, the client changes state on receipt of replies to requests. Note that some requests are effective at a future time or position (such as a PAUSE), and state also changes accordingly. If no explicit SETUP is required for the object (for example, it is

available via a multicast group), state begins at Ready. In this case, there are only two states, Ready and Playing. The client also changes state from Playing/Recording to Ready when the end of the requested range is reached.

The "next state" column indicates the state assumed after receiving a success response (2xx). If a request yields a status code of 3xx, the state becomes Init, and a status code of 4xx yields no change in state. Messages not listed for each state MUST NOT be issued by the client in that state, with the exception of messages not affecting state, as listed above. Receiving a REDIRECT from the server is equivalent to receiving a 3xx redirect status from the server.

state	message sent	next state after response
Init	SETUP	Ready
	TEARDOWN	Init
Ready	PLAY	Playing
	RECORD	Recording
	TEARDOWN	Init
	SETUP	Ready
Playing	PAUSE	Ready
	TEARDOWN	Init
	PLAY	Playing
	SETUP	Playing (changed transport)
Recording	PAUSE	Ready
	TEARDOWN	Init
	RECORD	Recording
	SETUP	Recording (changed transport)

A.2 Server State Machine

The server can assume the following states:

Init:
: The initial state, no valid SETUP has been received yet.

Ready:
: Last SETUP received was successful, reply sent or after playing, last PAUSE received was successful, reply sent.

Playing:
: Last PLAY received was successful, reply sent. Data is being sent.

Recording:
: The server is recording media data.

In general, the server changes state on receiving requests. If the server is in state Playing or Recording and in unicast mode, it MAY revert to Init and tear down the RTSP session if it has not received "wellness" information, such as RTCP reports or RTSP commands, from the client for a defined interval, with a default of one minute. The server can declare another timeout value in the Session response header (Section 12.37). If the server is in state Ready, it MAY revert to Init if it does not receive an RTSP request for an interval of more than one minute. Note that some requests (such as PAUSE) may be effective at a future time or position, and server state changes at the appropriate time. The server reverts from state Playing or Recording to state Ready at the end of the range requested by the client.

The REDIRECT message, when sent, is effective immediately unless it has a Range header specifying when the redirect is effective. In such a case, server state will also change at the appropriate time.

If no explicit SETUP is required for the object, the state starts at Ready and there are only two states, Ready and Playing.

The "next state" column indicates the state assumed after sending a success response (2xx). If a request results in a status code of 3xx, the state becomes Init. A status code of 4xx results in no change.

state	message received	next state
Init	SETUP	Ready
	TEARDOWN	Init
Ready	PLAY	Playing
	SETUP	Ready
	TEARDOWN	Init
	RECORD	Recording
Playing	PLAY	Playing
	PAUSE	Ready
	TEARDOWN	Init
	SETUP	Playing
Recording	RECORD	Recording
	PAUSE	Ready
	TEARDOWN	Init
	SETUP	Recording

Appendix B: Interaction with RTP

RTSP allows media clients to control selected, non-contiguous sections of media presentations, rendering those streams with an RTP media layer[24]. The media layer rendering the RTP stream should not be affected by jumps in NPT. Thus, both RTP sequence numbers and RTP timestamps MUST be continuous and monotonic across jumps of NPT.

As an example, assume a clock frequency of 8000 Hz, a packetization interval of 100 ms and an initial sequence number and timestamp of zero. First we play NPT 10 through 15, then skip ahead and play NPT 18 through 20. The first segment is presented as RTP packets with sequence numbers 0 through 49 and timestamp 0 through 39,200. The second segment consists of RTP packets with sequence number 50 through 69, with timestamps 40,000 through 55,200.

We cannot assume that the RTSP client can communicate with the RTP media agent, as the two may be independent processes. If the RTP timestamp shows the same gap as the NPT, the media agent will assume that there is a pause in the presentation. If the jump in NPT is large enough, the RTP timestamp may roll over and the media agent may believe later packets to be duplicates of packets just played out.

For certain datatypes, tight integration between the RTSP layer and the RTP layer will be necessary. This by no means precludes the above restriction. Combined RTSP/RTP media clients should use the RTP-Info field to determine whether incoming RTP packets were sent before or after a seek.

For continuous audio, the server SHOULD set the RTP marker bit at the beginning of serving a new PLAY request. This allows the client to perform playout delay adaptation.

For scaling (see Section 12.34), RTP timestamps should correspond to the playback timing. For example, when playing video recorded at 30 frames/second at a scale of two and speed (Section 12.35) of one, the server would drop every second frame to maintain and deliver video packets with the normal timestamp spacing of 3,000 per frame, but NPT would increase by 1/15 second for each video frame.

The client can maintain a correct display of NPT by noting the RTP timestamp value of the first packet arriving after repositioning. The sequence parameter of the RTP-Info (Section 12.33) header provides the first sequence number of the next segment.

Appendix C: Use of SDP for RTSP Session Descriptions

The Session Description Protocol (SDP, RFC 2327 [6]) may be used to describe streams or presentations in RTSP. Such usage is limited to specifying means of access and encoding(s) for:

aggregate control:
: A presentation composed of streams from one or more servers that are not available for aggregate control. Such a description is typically retrieved by HTTP or other non-RTSP means. However, they may be received with ANNOUNCE methods.

non-aggregate control:
: A presentation composed of multiple streams from a single server that are available for aggregate control. Such a description is typically returned in reply to a DESCRIBE request on a URL, or received in an ANNOUNCE method.

This appendix describes how an SDP file, retrieved, for example, through HTTP, determines the operation of an RTSP session. It also describes how a client should interpret SDP content returned in reply to a DESCRIBE request. SDP provides no mechanism by which a client can distinguish, without human guidance, between several media streams to be rendered simultaneously and a set of alternatives (e.g., two audio streams spoken in different languages).

C.1 Definitions

The terms "session-level", "media-level" and other key/attribute names and values used in this appendix are to be used as defined in SDP (RFC 2327 [6]):

C.1.1 Control URL

The "a=control:" attribute is used to convey the control URL. This attribute is used both for the session and media descriptions. If used for individual media, it indicates the URL to be used for controlling that particular media stream. If found at the session level, the attribute indicates the URL for aggregate control.

Example:

```
a=control:rtsp://example.com/foo
```

This attribute may contain either relative and absolute URLs, following the rules and conventions set out in RFC 1808 [25]. Implementations should look for a base URL in the following order:

1. The RTSP Content-Base field
2. The RTSP Content-Location field
3. The RTSP request URL

If this attribute contains only an asterisk (*), then the URL is treated as if it were an empty embedded URL, and thus inherits the entire base URL.

C.1.2 Media streams

The "m=" field is used to enumerate the streams. It is expected that all the specified streams will be rendered with appropriate synchronization. If the session is unicast, the port number serves as a recommendation from the server to the client; the client still has to include it in its SETUP request and may ignore this recommendation. If the server has no preference, it SHOULD set the port number value to zero.

Example:

```
m=audio 0 RTP/AVP 31
```

C.1.3 Payload type(s)

The payload type(s) are specified in the "m=" field. In case the payload type is a static payload type from RFC 1890 [1], no other information is required. In case it is a dynamic payload type, the media attribute "rtpmap" is used to specify what the media is. The "encoding name" within the "rtpmap" attribute may be one of those specified in RFC 1890 (Sections 5 and 6), or an experimental encoding with a "X-" prefix as specified in SDP (RFC 2327 [6]). Codec-specific parameters are not specified in this field, but rather in the "fmtp" attribute described below. Implementors seeking to register new encodings should follow the procedure in RFC 1890 [1]. If the media type is not suited to the RTP AV profile, then it is recommended that a new profile be created and the appropriate profile name be used in lieu of "RTP/AVP" in the "m=" field.

C.1.4 Format-specific parameters

Format-specific parameters are conveyed using the "fmtp" media attribute. The syntax of the "fmtp" attribute is specific to the encoding(s) that the attribute refers to. Note that the packetization interval is conveyed using the "ptime" attribute.

C.1.5 Range of presentation

The "a=range" attribute defines the total time range of the stored session. (The length of live sessions can be deduced from the "t" and "r" parameters.) Unless the presentation contains media streams of different durations, the range attribute is a session-level attribute. The unit is specified first, followed by the value range. The units and their values are as defined in Section 3.5, 3.6 and 3.7.

Examples:

```
a=range:npt=0-34.4368
a=range:clock=19971113T2115-19971113T2203
```

C.1.6 Time of availability

The "t=" field MUST contain suitable values for the start and stop times for both aggregate and non-aggregate stream control. With aggregate control, the server SHOULD indicate a stop time value for which it guarantees the description to be valid, and a start time that is equal to or before the time at which the DESCRIBE request was received. It MAY also indicate start and stop times of 0, meaning that the session is always available. With non-aggregate control, the values should reflect the actual period for which the session is available in keeping with SDP semantics, and not depend on other means (such as the life of the web page containing the description) for this purpose.

C.1.7 Connection Information

In SDP, the "c=" field contains the destination address for the media stream. However, for on-demand unicast streams and some multicast streams, the destination address is specified by the client via the SETUP request. Unless the media content has a fixed destination address, the "c=" field is to be set to a suitable null value. For addresses of type "IP4", this value is "0.0.0.0".

C.1.8 Entity Tag

The optional "a=etag" attribute identifies a version of the session description. It is opaque to the client. SETUP requests may include this identifier in the If-Match field (see section 12.22) to only allow session establishment if this attribute value still corresponds to that of the current description. The attribute value is opaque and may contain any character allowed within SDP attribute values.

Example:

```
a=etag:158bb3e7c7fd62ce67f12b533f06b83a
```

One could argue that the "o=" field provides identical functionality. However, it does so in a manner that would put constraints on servers that need to support multiple session description types other than SDP for the same piece of media content.

C.2 Aggregate Control Not Available

If a presentation does not support aggregate control and multiple media sections are specified, each section MUST have the control URL specified via the "a=control:" attribute.

Example:

```
v=0
o=- 2890844256 2890842807 IN IP4 204.34.34.32
s=I came from a web page
t=0 0
c=IN IP4 0.0.0.0
m=video 8002 RTP/AVP 31
a=control:rtsp://audio.com/movie.aud
m=audio 8004 RTP/AVP 3
a=control:rtsp://video.com/movie.vid
```

Note that the position of the control URL in the description implies that the client establishes separate RTSP control sessions to the servers audio.com and video.com.

It is recommended that an SDP file contains the complete media initialization information even if it is delivered to the media client through non-RTSP means. This is necessary as there is no mechanism to indicate that the client should request more detailed media stream information via DESCRIBE.

C.3 Aggregate Control Available

In this scenario, the server has multiple streams that can be controlled as a whole. In this case, there are both media-level "a=control:" attributes, which are used to specify the stream URLs, and a session-level "a=control:" attribute which is used as the request URL for aggregate control. If the media-level URL is relative, it is resolved to absolute URLs according to Section C.1.1 above.

If the presentation comprises only a single stream, the media-level "a=control:" attribute may be omitted altogether. However, if the presentation contains more than one stream, each media stream section MUST contain its own "a=control" attribute.

Example:

```
v=0
o=- 2890844256 2890842807 IN IP4 204.34.34.32
s=I contain
i=<more info>
t=0 0
c=IN IP4 0.0.0.0
a=control:rtsp://example.com/movie/
m=video 8002 RTP/AVP 31
a=control:trackID=1
m=audio 8004 RTP/AVP 3
a=control:trackID=2
```

In this example, the client is required to establish a single RTSP session to the server, and uses the URLs rtsp://example.com/movie/trackID=1 and rtsp://example.com/movie/trackID=2 to set up the video and audio streams, respectively. The URL rtsp://example.com/movie/ controls the whole movie.

Appendix D: Minimal RTSP implementation

D.1 Client

A client implementation MUST be able to do the following :

* Generate the following requests: SETUP, TEARDOWN, and one of PLAY (i.e., a minimal playback client) or RECORD (i.e., a minimal recording client). If RECORD is implemented, ANNOUNCE must be implemented as well.
* Include the following headers in requests: CSeq, Connection, Session, Transport. If ANNOUNCE is implemented, the capability to include headers Content-Language, Content-Encoding, Content-Length, and Content-Type should be as well.
* Parse and understand the following headers in responses: CSeq, Connection, Session, Transport, Content-Language, Content-Encoding, Content-Length, Content-Type. If RECORD is implemented, the Location header must be understood as well. RTP-compliant implementations should also implement RTP-Info.
* Understand the class of each error code received and notify the end-user, if one is present, of error codes in classes 4xx and 5xx. The notification requirement may be relaxed if the end-user explicitly does not want it for one or all status codes.
* Expect and respond to asynchronous requests from the server, such as ANNOUNCE. This does not necessarily mean that it should implement the ANNOUNCE method, merely that it MUST respond positively or negatively to any request received from the server.

Though not required, the following are highly recommended at the time of publication for practical interoperability with initial implementations and/or to be a "good citizen".

* Implement RTP/AVP/UDP as a valid transport.
* Inclusion of the User-Agent header.
* Understand SDP session descriptions as defined in Appendix C
* Accept media initialization formats (such as SDP) from standard input, command line, or other means appropriate to the operating environment to act as a "helper application" for other applications (such as web browsers).

There may be RTSP applications different from those initially envisioned by the contributors to the RTSP specification for which the requirements above do not make sense. Therefore, the recommendations above serve only as guidelines instead of strict requirements.

D.1.1 Basic Playback

To support on-demand playback of media streams, the client MUST additionally be able to do the following:

* generate the PAUSE request;
* implement the REDIRECT method, and the Location header.

D.1.2 Authentication-enabled

In order to access media presentations from RTSP servers that require authentication, the client MUST additionally be able to do the following:

* recognize the 401 status code;
* parse and include the WWW-Authenticate header;
* implement Basic Authentication and Digest Authentication.

D.2 Server

A minimal server implementation MUST be able to do the following:

* Implement the following methods: SETUP, TEARDOWN, OPTIONS and either PLAY (for a minimal playback server) or RECORD (for a minimal recording server). If RECORD is implemented, ANNOUNCE should be implemented as well.
* Include the following headers in responses: Connection, Content-Length, Content-Type, Content-Language, Content-Encoding, Transport, Public. The capability to include the Location header should be implemented if the RECORD method is. RTP-compliant implementations should also implement the RTP-Info field.
* Parse and respond appropriately to the following headers in requests: Connection, Session, Transport, Require.

Though not required, the following are highly recommended at the time of publication for practical interoperability with initial implementations and/or to be a "good citizen".

* Implement RTP/AVP/UDP as a valid transport.
* Inclusion of the Server header.
* Implement the DESCRIBE method.
* Generate SDP session descriptions as defined in Appendix C

There may be RTSP applications different from those initially envisioned by the contributors to the RTSP specification for which the requirements above do not make sense. Therefore, the recommendations above serve only as guidelines instead of strict requirements.

D.2.1 Basic Playback

To support on-demand playback of media streams, the server MUST additionally be able to do the following:

* Recognize the Range header, and return an error if seeking is not supported.
* Implement the PAUSE method.

In addition, in order to support commonly-accepted user interface features, the following are highly recommended for on-demand media servers:

* Include and parse the Range header, with NPT units. Implementation of SMPTE units is recommended.
* Include the length of the media presentation in the media initialization information.
* Include mappings from data-specific timestamps to NPT. When RTP is used, the rtptime portion of the RTP-Info field may be used to map RTP timestamps to NPT.

Client implementations may use the presence of length information to determine if the clip is seekable, and visibly disable seeking features for clips for which the length information is unavailable. A common use of the presentation length is to implement a "slider bar" which serves as both a progress indicator and a timeline positioning tool.

Mappings from RTP timestamps to NPT are necessary to ensure correct positioning of the slider bar.

D.2.2 Authentication-enabled

In order to correctly handle client authentication, the server MUST additionally be able to do the following:

* Generate the 401 status code when authentication is required for the resource.
* Parse and include the WWW-Authenticate header
* Implement Basic Authentication and Digest Authentication

Appendix E: Authors' Addresses

Henning Schulzrinne
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue
New York, NY 10027
USA

EMail: schulzrinne@cs.columbia.edu

Anup Rao
Netscape Communications Corp.
501 E. Middlefield Road
Mountain View, CA 94043
USA

EMail: anup@netscape.com

Robert Lanphier
RealNetworks
1111 Third Avenue Suite 2900
Seattle, WA 98101
USA

EMail: robla@real.com

Appendix F: Acknowledgements

This memo is based on the functionality of the original RTSP document submitted in October 96. It also borrows format and descriptions from HTTP/1.1.

This document has benefited greatly from the comments of all those participating in the MMUSIC-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

Rahul Agarwal, Torsten Braun, Brent Browning, Bruce Butterfield, Steve Casner, Francisco Cortes, Kelly Djahandari, Martin Dunsmuir, Eric Fleischman, Jay Geagan, Andy Grignon, V. Guruprasad, Peter Haight, Mark Handley, Brad Hefta-Gaub, John K. Ho, Philipp Hoschka, Anne Jones, Anders Klemets, Ruth Lang, Stephanie Leif, Jonathan Lennox, Eduardo F. Llach, Rob McCool, David Oran, Maria Papadopouli, Sujal Patel, Ema Patki, Alagu Periyannan, Igor Plotnikov, Pinaki Shah, David Singer, Jeff Smith, Alexander Sokolsky, Dale Stammen, and John Francis Stracke.

References

1 Schulzrinne, H., "RTP profile for audio and video conferences with minimal control", RFC 1890, January 1996.

2 Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext transfer protocol - HTTP/1.1", RFC 2068, January 1997.

3 Yergeau, F., Nicol, G., Adams, G., and M. Duerst, "Internationalization of the hypertext markup language", RFC 2070, January 1997.

4 Bradner, S., "Key words for use in RFCs to indicate requirement levels", BCP 14, RFC 2119, March 1997.

5 ISO/IEC, "Information technology - generic coding of moving pictures and associated audio information - part 6: extension for digital storage media and control," Draft International Standard ISO 13818-6, International Organization for Standardization ISO/IEC JTC1/SC29/WG11, Geneva, Switzerland, Nov. 1995.

6 Handley, M., and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

7 Franks, J., Hallam-Baker, P., and J. Hostetler, "An extension to HTTP: digest access authentication", RFC 2069, January 1997.

8 Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

9 Hinden, B. and C. Partridge, "Version 2 of the reliable data protocol (RDP)", RFC 1151, April 1990.

10 Postel, J., "Transmission control protocol", STD 7, RFC 793, September 1981.

11 H. Schulzrinne, "A comprehensive multimedia control architecture for the Internet," in Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), (St. Louis, Missouri), May 1997.

12 International Telecommunication Union, "Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, May 1996.

13 McMahon, P., "GSS-API authentication method for SOCKS version 5", RFC 1961, June 1996.

14 J. Miller, P. Resnick, and D. Singer, "Rating services and rating systems (and their machine readable descriptions)," Recommendation REC-PICS-services-961031, W3C (World Wide Web Consortium), Boston, Massachusetts, Oct. 1996.

15 J. Miller, T. Krauskopf, P. Resnick, and W. Treese, "PICS label distribution label syntax and communication protocols," Recommendation REC-PICS-labels-961031, W3C (World Wide Web Consortium), Boston, Massachusetts, Oct. 1996.

16 Crocker, D. and P. Overell, "Augmented BNF for syntax specifications: ABNF", RFC 2234, November 1997.

17 Braden, B., "Requirements for internet hosts - application and support", STD 3, RFC 1123, October 1989.

18 Elz, R., "A compact representation of IPv6 addresses", RFC 1924, April 1996.

19 Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform resource locators (URL)", RFC 1738, December 1994.

20 Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

22 Braden, B., "T/TCP - TCP extensions for transactions functional specification", RFC 1644, July 1994.

22 W. R. Stevens, TCP/IP illustrated: the implementation, vol. 2. Reading, Massachusetts: Addison-Wesley, 1994.

23 Schulzrinne, H., Casner, S., Frederick, R. and V. Jacobson, "RTP: a transport protocol for real-time applications", RFC 1889, January 1996.

24 Fielding, R., "Relative uniform resource locators", RFC 1808, June 1995.

Full Copyright Statement